



# Network of Excellence

Thematic Priority 2

## FP6 – IST- 511368

# HYCON

**Hybrid Control: Taming Heterogeneity and Complexity of Networked Embedded Systems**

**Starting date:** 15 September 2004 **Duration:** 4 years

| | | |
|---|---|---|
| **Deliverable number** | D3.6.3 | |
| **Title** | Revised hybrid system interchange format | |
| **Work package** | WP3 | |
| **Due date** | Month 35 | |
| **Actual submission date** | 14/09/2007 v1.0 | |
| **Organisation name(s) of lead contractor for this deliverable** | UNIDO | |
| **Author(s)** | D.A. van Beek | d.a.v.beek@tue.nl |
| | M.A. Reniers | m.a.reniers@tue.nl |
| | J.E. Rooda | j.e.rooda@tue.nl |
| | R.R.H. Schiffelers | r.r.h.schiffelers@tue.nl |
| **With the help of** | | |
| **Nature** | Report | |
| **Revision** | v1.0 | 14/09/2007 15:34 |

**Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)**

**Dissemination Level**

| | | |
|---|---|---|
| PU | Public | **X** |
| PP | Restricted to other programme participants (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium (including the Commission Services) | |
| CO | Confidential, only for members of the consortium (including the Commission Services) | |

| | |
|---|---|
| **Executive summary** | The compositional interchange format for hybrid systems (CIF) is syntactically and semantically defined in terms of an interchange automaton in an abstract format, allowing among others differential algebraic equations (DAEs), including fully implicit or switched DAEs, discrete, continuous and algebraic variables that can be internal or external, and `time can progress predicates'. The following operators are provided: operators for parallel composition, including synchronization by means of shared labels, and communication by means of shared variables and CSP channels, and operators for action hiding, variable hiding and urgent actions. The semantics of the CIF is formally defined in terms of a hybrid transition system. This allows development of transformations to and from other formalisms that can be proven to preserve essential properties, and it allows a clear separation between the mathematical meaning of a model and implementation aspects such as algorithms used for solving differential algebraic equations. A concrete format is defined for modeling. Its semantics is defined by means of a translation to the abstract format. The concrete format adds inputs, outputs and open and closed scopes to enable modular and hierarchical specifications. The concrete format is illustrated by means of a bottle filling line example. |

# Contents

# 1 Introduction

Our intention is to establish inter-operability of a wide range of tools by means of model transformations to and from a compositional interchange format (CIF) that is defined in terms of an *interchange automaton*. The application domain of the interchange automaton format consists of languages and tools from computer science and from dynamics and control for modeling, simulation, analysis, controller synthesis, and verification in the area of hybrid and timed systems. The purpose of the CIF is twofold: First, to establish inter-operability of a wide range of tools by means of model transformations to and from the CIF, see Figure 1 for an example. In this way, the implementation of many bi-lateral translators between specific formalisms can be avoided as shown in Figures 2 and 3. Second, to provide a generic modeling formalism and simulator for a wide range of hybrid systems, see Figures 4 for an example.

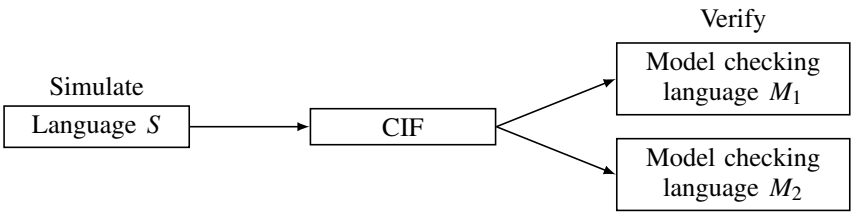


Figure 1: Using CIF transformations for simulation and verification.

Our main requirements for the interchange format are summarized below. A more detailed discussion of these requirements follows in Sections 2 and 3.

1. It should have a formal and compositional semantics, based on (hybrid) transition systems, and allow property preserving model transformations.
2. Its concepts should be based on mathematics, and independent of implementation aspects such as equation sorting, and numerical equation solving algorithms.

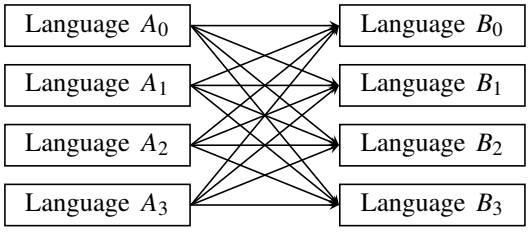


Figure 2: Multiple model transformations without the CIF.

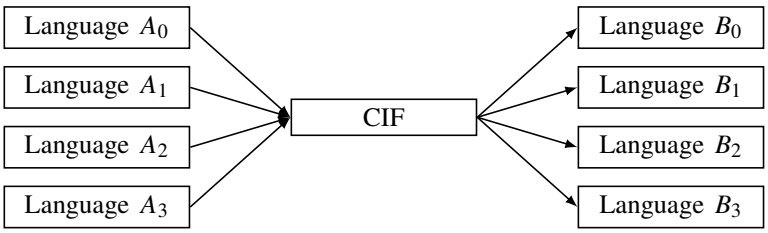


Figure 3: Multiple model transformations using the CIF.

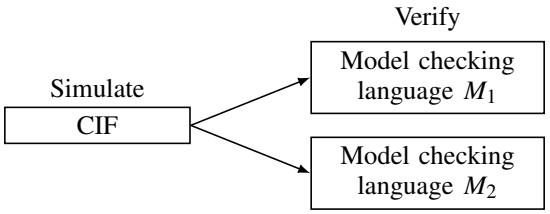


Figure 4: Modeling directly in the CIF.

3. It should support arbitrary differential algebraic equations (DAEs), including fully implicit equations, higher index systems, algebraic loops, steady state initialization, switched systems such as piecewise affine systems, and DAEs with discontinuous right hand sides.
4. It should support a wide range of urgency concepts, such as used in hybrid automata, including 'urgency predicates', 'deadline predicates', 'triggering guard semantics', and 'urgent actions'.
5. It should support parallel composition with synchronization by means of shared variables, shared actions, and CSP channels [26].
6. It should support hierarchy and modularity to allow the definition of parallel modules and modules that can contain other modules (hierarchy), and to allow the definition of variables and actions as being local to a module, or shared between modules.

Other work on interchange formats for hybrid systems has been carried out in different projects: in the MoBIES project, the Hybrid System Interchange Format (HSIF) [33] is defined; in [36] an abstract semantics of an interchange format based on the Metropolis meta model is defined (this work is a continuation of the COLUMBUS project [14]); and in the HYCON NoE [27], an interchange format for switched linear systems [13] in the form of piecewise affine systems (PWAs) is defined.

In HSIF, a network of hybrid automata is used for model representation. The network behaves as a parallel composition of its automata, without hierarchy or modules. Variables can be shared or local, and the communication mechanism is based on broadcasting of boolean 'signals', where signals are partitioned in input and output signals. Each signal is required to be either a global input to the network or to be modified by exactly one automaton. The semantics is defined only for 'acyclic dependency graphs' with respect to the use of signals. The time dependent behavior is specified by means of ordinary differential equations (ODEs), together with algebraic relations of the form $x = f(x_1, \ldots, x_n)$, and invariants. The equation $\dot{x} = 0$ is assumed for each shared variable. Circular dependencies of the algebraic equations, i.e. algebraic loops, are not allowed. To provide for 'synchronous execution' of discrete steps, each automaton has a default transition. There are no algebraic variables, and there is no form of urgency predicate or urgent action. Execution of the network is defined as an alternating sequence of continuous steps followed by discrete steps, where each discrete step of the network is the result of a sequence of discrete steps of each of the automata taken in an order that respects automata dependencies [33]. The interchange automaton format defined in this report aims to be more general than HSIF, and does not incorporate tool limitations, such as restrictions on circular dependencies, or restrictions on shared variables or algebraic loops, in its compositional formal semantics.

The abstract semantics presented in [36], takes implementation considerations into account, such as equation sorting, iterations that may be required for state-event detection, and iterations for reaching a fixed-point in case of algebraic loops. The semantics is defined in terms of functions and algorithms such as init, markchange, and solve. This is different from the compositional formal semantics as defined in Section 5, which aims at defining the *mathematical* meaning of interchange automata, independently of implementation aspects such as equation sorting or state-event detection. For example, the semantics defines the mathematical meaning of a switched

system of equations, such as a PWA system, but an implementation may choose to implement such switching behavior with or without state-event detection.

A transformation from the PWA-based interchange format [13] to the interchange automaton format will be developed. Based on this transformation, several tools, based on among others PWA, HYSDEL, MLD (see [22] for an overview relating these languages) can then be connected to the interchange automaton format.

In principle, an interchange automaton can be specified in three different formats: First, an abstract format as defined in Section 4 of this report. The purpose of this format is to facilitate definition of the formal semantics. Second, a concrete format. The purpose of this format is to provide user friendly syntax, that can be used for modeling directly in the interchange automaton format. Third, a transfer format. The purpose of this format is to facilitate the file generation and parsing process. This format can, for example, be specified in an extensible markup language (XML) format, that is supported by means of libraries in many different languages.

The remainder of this article is organized as follows: Section 2 discusses the importance of a compositional formal semantics, Section 3 discusses the concepts present in the interchange automaton format, Sections 4 and 5 define the syntax and semantics of interchange automata, respectively, Section 6 shows how the operators of interchange automata can be eliminated resulting in atomic interchange automata, Sections 7 and 9 define the concrete syntax and its mapping to the abstract syntax, respectively, Section 8 presents a bottle filling system example and Section 10 presents concluding remarks.

# 2 Importance of a compositional formal semantics

A formal semantics defines the mathematical meaning of a valid model given in the interchange automaton format, i.e., it defines all possible behaviors of the model for certain initial conditions. A formal semantics allows a clear distinction between the mathematical meaning of a model and implementation aspects. It defines a standard against which implementations, such as simulation or verification implementations can be evaluated. To use the interchange automaton format for verification purposes, translations from models to the interchange format, and vice versa, should preserve essential properties. I.e., verification results obtained for a derived model should also be valid for the original model specified in another language.

The different languages may have different features and semantics, complicating translations between them. To keep the translations between the interchange automaton format and the other languages manageable, in terms of complexity, it is important that transformations between parts of specifications within the interchange format itself can be defined. To allow such transformations, it is essential that the semantics of the interchange automaton format is *compositional*. I.e., that the notion of equivalence is a congruence for all operators of the interchange automaton format, see Section 5.3. Parts of a model can then be replaced by equivalent parts without changing the meaning of the model.

Consider, for instance, a transformation of a model in a simulation language, such as Modelica [41] or EcosimPro [15], to a verification tool, such as PHAVER [18] or HYTECH [25]. The simulation languages use a triggering guard semantics (see Section 3.4), whereas the verification tools use invariants to force switching to a different location. Defining direct translations from the simulation languages to the verification tools and reasoning about the correctness of the translations would be difficult, since the simulation languages do not have a formal semantics and the urgent guards in two languages would need to be transformed into invariants in combination with non-urgent guards in two other languages. By using the compositional interchange format as an intermediate, the complicated direct translations can be replaced by more straightforward translations from the simulation languages to the interchange format, using urgent guards, and from the interchange format to the verification tools, using invariants; in combination with a transformation in the interchange format from urgent guards to invariants.

For simulation of hybrid systems based on nonlinear DAEs, usually numerical solvers are used [12]. Numerical solvers can be parameterized with, among others, different settings for the relative and/or absolute accuracies. This means that simulation of the same model using different simulators, or using different solvers or different parameter settings for the same simulator, will usually lead to different results. In such cases, there will obviously be a difference between the formal semantics, which mathematically defines the allowed behaviors of the model, and the result of a simulation run, which gives a numeric approximation of one of the allowed behaviors. The formal semantics then defines the 'reference behavior' which can be used to evaluate the quality of the numeric approximation provided by a simulator.

The fact that a formal semantics defines the mathematical meaning of a model is also useful when dealing with advanced simulators for complex nonlinear DAEs. Several techniques exist for efficient simulation of such DAEs, such as ordering of the equations in block lower triangular structure, tearing, and reduction of the number of equations and unknowns by symbolic manipulation and solving [32]. Clearly, these techniques for improving simulation efficiency should not affect the simulation results. A formal semantics unambiguously defines the mathematical meaning of a model, and is thus completely independent of techniques used by advanced simulation algorithms.

# 3 Concepts in the interchange automaton format

## 3.1 Differential algebraic equations

Modeling of physical systems, such as mechanical or chemical systems frequently leads to DAEs. Algebraic constraints can also be the result of stateless components such as proportional controllers. DAEs can be modeled and simulated using languages such as Modelica and EcosimPro. DAEs can be specified in the invariants of an interchange automaton, since such invariants are predicates over all variables, including the dotted variables. Flow clauses are supported for reasons of compatibility with existing hybrid automata. The reason for not enforcing a separation between invariants (over non-dotted variables) and flow clauses (over dotted variables), as in existing hybrid automata, is that such a separation is absent in the mathematical theory of dynamical systems, including control theory. In many cases, fully implicit DAEs, such as $\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t) = \mathbf{0}$, cannot even be rewritten to a form where the algebraic constraints and the differential constraints are separated, such as the semi-explicit form $\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, \mathbf{y}, t)$, $\mathbf{h}(\mathbf{x}, \mathbf{y}, t) = \mathbf{0}$, where $\mathbf{x}$ and $\mathbf{y}$ are the continuous and algebraic variables, respectively. The generalized invariant allows us to consider the four expressions $x = 1 \wedge x = 2$, $\dot{x} = 1 \wedge \dot{x} = 2$, $\dot{x} = y \wedge \dot{x} = 2y \wedge y = 1$ and 'false' to be equivalent (bisimilar): no behavior is possible.

The initialization clause of the interchange automaton is also defined as a predicate over all variables, including the dotted variables. This allows more general initializations than usually allowed in hybrid automata. In particular, steady state initialization, as available in Modelica and EcosimPro, is supported. E.g. by defining $\dot{\mathbf{x}} = \mathbf{0}$ as initialization predicate for a location with invariant $\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t) = \mathbf{0}$, the initial state is defined as the 'steady state', that is the solution of the set of DAEs such that all derivatives are zero: $\mathbf{f}(\mathbf{0}, \mathbf{x}, \mathbf{y}, t) = \mathbf{0}$.

## 3.2 Discrete, continuous and algebraic variables

The interchange automaton defines three classes of variables: the discrete and continuous variables, and in addition the algebraic variables. The differences are as follows: First, continuous variables are the only variables for which dotted variables (derivatives) can be used in models. Second, the values of discrete variables remain constant when model time progresses, the values of continuous variables may change according to a continuous function of time when model time progresses, and the values of algebraic variables may change according to a discontinuous function of time. Third, the values of the discrete and continuous variables do not change in action transitions unless such changes are explicitly specified, for example by assigning a new value

to such a variable. The values of algebraic variables can change arbitrarily in action transitions, unless such changes are explicitly restricted, for example by assigning a new value to such a variable. Fourth, there is a difference between the different classes of variables with respect to how the resulting values of the variables in a transition relate to the starting values of the variables in the next transition. The resulting value of a discrete or continuous variable in a transition always equals its starting value in the next transition. For algebraic variables there is no such relation, because algebraic variables are not part of the state. This means that the starting value of the trajectory (the values of a variable as a function of time when time passes) of a discrete or continuous variable equals its value in the state. The starting value of the trajectory of an algebraic or dotted variable can be any value that is allowed by the invariant and flow condition (and possible also the initial condition) of the automaton.

The state of an interchange automaton consists of, among others, the interchange automaton itself, and a valuation of the discrete and continuous variables (see Section 5 for a more precise definition of the state). The values of the dotted variables and the algebraic variables are not contained in the state. The reason for this is that the state of an interchange automaton represents all information needed to determine future behavior, i.e., the state of a system makes the system's history irrelevant. The dotted and algebraic variables are not needed in the state, because their values are determined by the predicates of the interchange automaton: in particular by the initial conditions, the flow conditions, the invariants and the jump predicates as defined in Section 4.

The different semantics of the variable classes lead to different use. In most models, the values of discrete variables are defined by assignments, whereas the values of algebraic variables are defined by invariants (equations). The values of continuous variables are usually defined by invariants or flow conditions, in combination with assignments for (re)-initialization.

In most languages that allow (implicit) DAEs, such as Modelica [41], EcosimPro [15], and Simulink [40], the distinction between continuous and algebraic variables is implicitly made by considering all continuous variables that do not occur differentiated as algebraic. EcosimPro also implicitly makes the distinction between discrete and continuous variables: variables of type real that occur differentiated are continuous (unless the variable is prefixed by the DISCR qualifier, in which case it is discrete). Variables of other types (such as integer, boolean and string) are discrete.

### 3.3 Automata related concepts

Many different hybrid automaton definitions exist. Some definitions require solutions for the continuous variables to be differentiable functions, e.g. in [24, 3]. Other definitions allow the more general case of piecewise differentiable or piecewise continuous functions, e.g. in [39]. Such restrictions can be realized in the interchange automaton format by means of the parameters $F$ and $G$ as defined in Section 5. In [30], for each variable a 'dynamic type' can be defined. However, since we did not find such expressivity in tools, the interchange automaton format allows the definition of the dynamic type for the algebraic and continuous variable classes, not for each individual variable.

With respect to the meaning of jump predicates, that define the behavior of the variables in action transitions, differences also occur: in [24] the variables can in principle perform arbitrary jumps unless restricted by the jump predicate, in [25], variables in principle remain unchanged unless changes are enforced by the jump predicate by means of primed variables. The first behavior is obtained by an interchange automaton that defines the set of jumping variables $W$ (see Sections 4 and 5.1) at each edge to be equal to the set of all variables. The second kind of behavior is obtained by defining the set $W$ as the union of all primed variables of the jump predicate. The specification of a set of jumping variables and a jump predicate for each edge of an interchange automaton is based on [3].

The interchange automaton format is expressive enough to deal with verification tools such as PHAVER [18] and HYTECH [25]. The behavior of the algebraic variables from the interchange automaton is related to the external variables from the semantical hybrid I/O automaton defined

in [30]. In this I/O automaton, the external variables are also not part of the state, and they can have a dynamic type that allows discontinuous trajectories. The state is defined by the values of the internal variables, and discrete transitions (action transitions) are defined only on internal variables. The interchange automaton format can express this as a special case, since the different classes of variables, action transitions, and hiding/abstraction are orthogonal concepts in the interchange automaton format.

The basic elements of hierarchy and modularity that are supported by the interchange format are parallel composition, and hiding of variables and actions. Interchange automata can be grouped by means of parallel composition, and variables and/or actions that are meant to be local to that group can be hidden from the environment of the group. The concrete interchange format, that will be developed, will define modularity in terms of the basis elements of the abstract format.

### 3.4 Urgency

The concept of urgency allows the passing of time up to a certain point. There are essentially two kinds of urgency:

1. Urgency that is defined for an atomic automaton by means of one or more predicates. Such predicates can be associated to a location, or to outgoing edges of the location.

2. Urgency that is defined as an operation on a composition of one or more automata. Such an operation defines a set of actions as urgent for the composition. The operation allows the passing of time up to the point when one or more of the urgent actions can be executed.

By means of urgency, the execution of actions can be given priority over the passing of time. Urgency, however, is not used to give the execution of "urgent" actions priority over "non-urgent" actions. Urgency only restricts the passing of time until a certain point of time. When such a point of time is reached, in principle all enabled actions can be executed.

The first kind of urgency is defined in many different forms. The `tcp` (*time can progress*) predicate [35], is a predicate over the variables of the automaton and time. The predicate is associated to a location. It allows passing of time in a location for as long as the predicate is true. Related to the `tcp` predicate is the *stopping condition* [19], which is a predicate on the variables of the automaton, also associated to a location, and which allows passing of time in a location for as long as the stopping condition is false, or in other words, until the time-point when the stopping condition is true. *Deadline predicates* [11] and *urgency predicates* [19] are associated to the edges of an automaton. Deadline predicates allow passing of time in a location until the time-point that one or more deadline predicates of the outgoing edges of the location become true. Whenever a deadline predicate of an edge becomes true, the guard associated to that edge must also be true: the deadline predicate must imply the guard. Urgency predicates are similar to deadline predicates; the only difference is that they do not have the restriction that the urgency predicate should imply the guard. Urgency predicates allow passing of time in a location until the point of time that for one or more of the outgoing edges, the guard and the urgency predicate are both true.

Restricting a `tcp` predicate as a predicate over the variables of an automaton makes it equal to the negation of a stopping condition. Deadline predicates and urgency predicates are less expressive. They can both be expressed in terms of stopping conditions, see [19], or as `tcp` predicates. E.g. the stopping condition of a location corresponds to the disjunction of all deadline predicates of the outgoing edges of the location. Note that a flow condition which is false in a hybrid automaton is equivalent to a stopping condition that is true, or a `tcp` predicate that is false. The interchange automaton format adopts the `tcp` predicate.

In simulation languages, such as Modelica, EcosimPro, and HyVisual, usually a triggering or urgent guard semantics is used, meaning that the passing of time in a location is allowed until the time-point that any of the guards of the outgoing edges becomes true. This is equivalent to a stopping condition associated to the location that is the disjunction of the guards of all outgoing

edges, or to a `tcp` predicate associated to the location that is the conjunction of the negated guards of all outgoing edges.

The second kind of urgency, as for example defined in [10] and [5], is often available with restrictions only. E.g. in HYTECH, edges can be defined as urgent. The composition of urgent actions is required to be well-formed: 'whenever two components synchronize on a label, if one transition is urgent then the other must either be urgent, or have a jump condition expressible as a guarded command with its guard being either the predicate true or the predicate false' [25]. A second restriction is that 'if there exists an urgent transition from a location $v$ to a location $v'$, then for all valuations satisfying the invariant of $v$, an urgent transition to $v'$ should exist'. The timed automaton language UPPAAL [29] appears to define urgent communication by allowing the definition of channels either as urgent or non-urgent, but the semantics can be defined in terms of stopping conditions: time can pass in a location for as long as 1) none of its edges synchronize via an urgent channel, and 2) for all edges that synchronize via an urgent channel the guards remain false [42].

The interchange automaton format defines the second kind of urgency by means of the urgent action operator. Note that defining this kind of urgency by means of labeling certain edges or actions as urgent may lead to bisimulation not being a congruence for parallel composition, as described in [9]. Another example is the ASAP flag that can be attached to an edge in HYTECH. In such cases, replacing a part of a specification by another part with the same behavior may lead to different behavior of the complete system. Straightforward translations of such languages to and from the interchange format is in principle possible if each action (or edge with a certain action) is either always urgent or never urgent in a model. If the same action (or edge with a certain action) occurs both as urgent and not urgent, it may be necessary to eliminate parallel composition, as for example described in Section 6.1, before translation to the interchange format is possible.

### 3.5 Synchronous systems related concepts

An overview of the synchronous approach, as adopted by the three synchronous languages Esterel, Lustre and Signal, is given in [8]. Essential to this approach is the division of time into discrete instants and the distinction of inputs and outputs of a system. Execution of a synchronous model is, in principle, a deterministic transformation, at each time-instant, of the values for each of the inputs and internal state, to the values of the outputs and the internal state. A characteristic difference with hybrid automaton related formalisms is the semantics of parallel composition. Hybrid automaton related formalisms usually have an interleaving, non-deterministic semantics for the execution of actions. Synchronous languages in principle define the behavior of parallel composition of input/output systems as the (deterministic) conjunction of the behaviors. In practice, however, such synchronous behavior is difficult to achieve, and many different semantics exist, ranging from true synchronous behavior to full interleaving. Interaction in synchronous systems is usually based on shared variables and/or 'signal broadcasting'. In [28], a formal semantics of timed and hybrid statecharts is presented. Signal broadcasting is formally defined by means of 'volatile variables'. In Harel's Statecharts [21], the semantics is defined in terms of 'steps' and 'supersteps'. In other languages these may be referred to as 'microsteps' and 'steps'. Many, possibly conflicting, definitions of microsteps based semantics exist [8]. The comparative study [43] discusses several issues of the Statechart semantics and treats more than twenty Statechart variants. Other languages that have been influenced by synchronous systems are Masaccio [23] and, recently also Charon [4]. Because of the many different ways of dealing with 'synchronous behavior', the synchronous approach needs to be studied in more detail before incorporating it in the interchange automaton.

The synchronous approach is to some extent already present in the interchange automaton, since the parallel composition of interchange automata takes as the behavior for time transitions the conjunction of the behaviors of the individual automata. Furthermore, the interchange automaton shares the *substitution principle* [20] with Lustre: an equation $x = e$, where $x$ is a variable and $e$ an expression, defined in the invariant of a single location automaton without edges, allows the

substitution of $x$ by its defining expression $e$ everywhere in automata that are placed in parallel (assuming that the automata share variable $x$ and the variables of expression $e$). This is referred to as the *consistent equation semantics* in [5]. Furthermore, each connection between an input and output of synchronous models, including Simulink [40] blocks, can be modeled by means of an algebraic variable in an interchange automaton. The function $f$ of the block, say $\mathbf{y} = f(\mathbf{x}, \mathbf{u})$, where $\mathbf{x}$ is the state, $\mathbf{y}$ is the output, and $\mathbf{u}$ is the input, is modeled by means of an equation in the invariant of the interchange automaton.

# 4 Abstract syntax of interchange automata

**Notation 4.1.** The following notations are defined:

- A set $\mathcal{V}$ of variables, a set of basic action labels $\mathcal{L}_{\text{basic}}$, which does not include the predefined non-synchronizing action $\tau$, a set of channel labels $\mathcal{H}$, and a set of values $\Lambda$ are assumed. The set $\mathcal{L}_{\text{com}}$ denotes the set of CSP action labels. It is defined as $\mathcal{L}_{\text{com}} = \{h!cs, h?cs, h!?cs \mid h \in \mathcal{H}, cs \in \Lambda^*\}$, where $h \in \mathcal{H}$ denotes a channel, and $cs \in \Lambda^*$ denotes a list $[c_1, \ldots, c_n]$ of values ($c_i \in \Lambda$, $1 \leq i \leq n$). The CSP actions labels $h!cs$, $h?cs$, $h!?cs$ are called send action label, receive action label, synchronization action label (when $cs$ is the empty list) and communication action label (when $cs$ is a non-empty list), respectively. We assume the set of basic action labels and the set of CSP action labels to be disjoint: $\mathcal{L}_{\text{basic}} \cap \mathcal{L}_{\text{com}} = \emptyset$. The set $\mathcal{L}$ denotes the set of basic and CSP action labels $\mathcal{L}_{\text{basic}} \cup \mathcal{L}_{\text{com}}$, and the set $\mathcal{L}_\tau$ denotes the set $\mathcal{L} \cup \{\tau\}$.
- For a set of variables $S \subseteq \mathcal{V}$, $\dot{S} = \{\dot{x} \mid x \in S\}$ denotes the set of dotted variables.
- For a set of variables $S \subseteq \mathcal{V}$, $\text{Pred}(S)$ denotes the set of all predicates over variables from $S$, and $\text{Expr}(S)$ denotes the set of all expressions over variables from $S$.
- $f : A \mapsto B$ and $g : A \to B$ define a partial function $f$ and a total function $g$, both with domain $A$ and codomain $B$.

**Definition 4.2** (Atomic Interchange Automaton)**.** An *atomic interchange automaton* is a tuple $(X, X_{\text{i}}, \text{dtype}, V, v_0, \text{init}, \text{flow}, \text{inv}, \text{tcp}, L, E)$ where

- $X \subseteq \mathcal{V}$ is a finite set of variables, $X_{\text{i}} \subseteq X$ is the set of *internal* variables, and $X_{\text{e}} = X \setminus X_{\text{i}}$ is the set of *external* variables.
- $\text{dtype} : X \to \{\text{disc}, \text{cont}, \text{alg}\}$ is a function that associates to each variable a dynamic type: *discrete*, *continuous* or *algebraic*. The sets $X_{\text{disc}}, X_{\text{cont}}, X_{\text{alg}}$ are defined as $X_t = \{x \in X \mid \text{dtype}(x) = t\}$ for $t \in \{\text{disc}, \text{cont}, \text{alg}\}$, and $X_{\text{state}} = X_{\text{disc}} \cup X_{\text{cont}}$ is the set of *state* variables.
- $V$ is a finite non-empty set of *vertices*, called *locations*, and $v_0 \in V$ is the initial location.
- $\text{init} \in \text{Pred}(\tilde{X})$ is the initial condition. For $Y \subseteq X$, $\tilde{Y} = Y \cup \{\dot{y} \mid y \in Y \cap X_{\text{cont}}\}$ is the extension of $Y$ with the dotted versions of the continuous variables in $Y$.
- $\text{flow}, \text{inv}, \text{tcp} : V \to \text{Pred}(\tilde{X})$, are functions that each associate to each location $v \in V$ a predicate describing the *flow condition*, the *invariant*, and the *urgency condition*, respectively.
- $L \subseteq \mathcal{L}_{\text{basic}}$ is a finite set of synchronizing action labels. Usually, this set includes at least the labels that occur on the edges of the automaton, in which case the set $L$ is referred to as the *alphabet* of the automaton.
- $E = V \times \text{Pred}(\tilde{X}) \times (\mathcal{L}_{\text{basic}} \cup \{\tau\} \cup C_X) \times (\mathcal{P}(\tilde{X}) \times \text{Pred}(\tilde{X} \cup \tilde{X}^-)) \times V$ is a finite set of *edges*, such that for each element $(v, g, a, (W, r), v') \in E$, $v$ and $v'$ are the *source* and *target*

locations, respectively, $g$ is the *guard*, $a$ is an *action statement*, $W \subseteq \tilde{X}$ is a set of jumping variables (the value of which may change as a result of an action transition), and $r$ is the *jump predicate*, also called *reset map*. For any $Y \subseteq (\mathcal{V} \cup \dot{\mathcal{V}})$, $Y^- = \{y^- \mid y \in Y\}$ denotes the set of minus superscripted variables that represent the values of variables before an action transition. Three kinds of action statements exist: basic action labels $a \in L$ that synchronize on the basis of equality, the predefined non-synchronizing $\tau$ action, and CSP statements $a \in C_X$, where $C_X = \{\ h!\mathbf{e},\ h?\mathbf{x},\ h!?,\ h!?\mathbf{x} := \mathbf{e} \mid h \in \mathcal{H},\ \{\mathbf{e}\} \subseteq \text{Expr}(\tilde{X}),\ \{\mathbf{x}\} \subseteq \tilde{X}\ \}$, where $\mathbf{e}$ and $\mathbf{x}$ are either empty ($n = 0$) or denote comma separated sequences $e_1, \ldots, e_n$ and $x_1, \ldots, x_n$ of expressions and variables ($n \geq 1$), respectively. The CSP statements $h!\mathbf{e}$, $h?\mathbf{x}$, $h!?$, $h!?\mathbf{x} := \mathbf{e}$ are called send statement, receive statement, synchronization statement, and communication statement, respectively. The CSP synchronization statement $h!?$ and communication statement $h!?x := e$ are the result of elimination of parallel composition as defined in Section 6.1. They do not normally occur in models. We assume the set of CSP statements to be disjoint from the set of basic action labels: $C_X \cap \mathcal{L}_{\text{basic}} = \emptyset$. This requirement is used in the section on elimination of parallel composition.

For most automaton based formalisms, the alphabet of an automaton must include at least all basic action labels that are used in the automaton. This was reflected in the previous definition of the CIF in [7, 6], where the set of edges $E$ was defined as $E = V \times \text{Pred}(\tilde{X}) \times (L \cup \{\tau\}) \times (\mathcal{P}(\tilde{X}) \times \text{Pred}(\tilde{X} \cup \tilde{X}^-)) \times V$. The current definition is more general; apart from the fact that it allows CSP channels, it also allows basic action labels $a \in \mathcal{L}_{\text{basic}} \cup \{\tau\}$ on edges. Thus, it allows the use of action labels on edges that are not in the alphabet of the automaton. Such actions do not synchronize with actions of other automata of a parallel composition (see the semantics of parallel composition in Section 5.2). This extension of the CIF is a conservative extension. That is, automata that use only synchronizing actions, as in the previous definition of the CIF, have the same synchronization behavior in the new semantics.

Note that the *dynamic* type of a variable gives information about its time-dependent behavior. E.g. the value of a discrete variable remains constant when time passes, whereas the value of a continuous variable changes as a continuous function of time. The *static* type, such as real, integer or boolean, gives information about the domain in which the variable takes values.

The interchange automaton format consists of automata, and operators for parallel composition, for hiding of actions and variables, and for the definition of urgent actions. The automata and operators can be freely combined:

**Definition 4.3** (Interchange automaton)**.** The set of interchange automata $\mathcal{A}$ is defined by the following grammar for the interchange automata $\alpha \in \mathcal{A}$:

| | | | |
|---|---|---|---|
| $\alpha$ | ::= | $\alpha_{\text{atom}}$ | atomic interchange automaton |
| | \| | $\alpha \parallel \alpha$ | parallel composition |
| | \| | $\text{hidevar}_{X_\text{h}}(\alpha, \sigma_\text{h})$ | variable hiding operator |
| | \| | $\text{hideact}_{L_\text{h}}(\alpha)$ | action hiding operator |
| | \| | $\text{urgent}_{L_\text{u}}(\alpha)$ | urgent action operator |
| | \| | $\text{encap}_{L_\text{e}}(\alpha)$ | action encapsulation operator, |

where

- $\alpha_{\text{atom}}$ denotes an atomic interchange automaton;
- $X_\text{h} \subseteq \mathcal{V}$ denotes a set of variables to hide and $\sigma_\text{h} : X_\text{h} \mapsto \Lambda$ denotes a (partial) valuation for the hidden state variables of interchange automaton $\alpha$;
- $L_\text{h} \subseteq \mathcal{L}$ denotes a set of actions to hide;
- $L_\text{u} \subseteq \mathcal{L}_\tau$ denotes a set of nondelayable actions. If such an action can be executed by the automaton $\alpha$, the automaton is not allowed to delay;

- $L_e \subseteq \mathcal{L}$ denotes a set of actions that are blocked. The main purpose of the encapsulation operator is to enable blocking of send and receive actions via channels, and thus to restrict CSP actions to synchronization and communication.

In the next sections, three auxiliary functions on interchange automata are used. These are defined below. The functions $\text{var}_e$, $\text{var}_{e,st}$ and act extract the sets of external variables, external state variables, and external (non-hidden) basic action labels[1], respectively, from an interchange automaton:

**Definition 4.4.** For $\alpha_a = (X, X_i, \text{dtype}, V, v, \text{init}, \text{flow}, \text{inv}, \text{tcp}, L, E)$, and $\alpha, \alpha_1, \alpha_2 \in \mathcal{A}$ we define the functions $\text{var}_e, \text{var}_{e,st} : \mathcal{A} \to \mathcal{P}(\mathcal{V})$ and $\text{act} : \mathcal{A} \to \mathcal{P}(\mathcal{L}_{\text{basic}})$ as follows:

$$
\begin{aligned}
&\text{var}_{e,st}(\alpha_a) = X_{\text{state}} \cap X_e \\
&\text{var}_{e,st}(\alpha_1 \parallel \alpha_2) = \text{var}_{e,st}(\alpha_1) \cup \text{var}_{e,st}(\alpha_2) \\
&\text{var}_{e,st}(\text{hideact}_{L_h}(\alpha)) = \text{var}_{e,st}(\alpha) \\
&\text{var}_{e,st}(\text{hidevar}_{X_h}(\alpha, \sigma_h)) = \text{var}_{e,st}(\alpha) \setminus X_h \\
&\text{var}_{e,st}(\text{urgent}_{L_u}(\alpha)) = \text{var}_{e,st}(\alpha) \\
&\text{var}_{e,st}(\text{encap}_{L_e}(\alpha)) = \text{var}_{e,st}(\alpha)
\end{aligned}
$$

| | |
|---|---|
| $\text{var}_e(\alpha_a) = X_e$ | $\text{act}(\alpha_a) = L$ |
| $\text{var}_e(\alpha_1 \parallel \alpha_2) = \text{var}_e(\alpha_1) \cup \text{var}_e(\alpha_2)$ | $\text{act}(\alpha_1 \parallel \alpha_2) = \text{act}(\alpha_1) \cup \text{act}(\alpha_2)$ |
| $\text{var}_e(\text{hideact}_{L_h}(\alpha)) = \text{var}_e(\alpha)$ | $\text{act}(\text{hideact}_{L_h}(\alpha)) = \text{act}(\alpha) \setminus L_h$ |
| $\text{var}_e(\text{hidevar}_{X_h}(\alpha, \sigma_h)) = \text{var}_e(\alpha) \setminus X_h$ | $\text{act}(\text{hidevar}_{X_h}(\alpha, \sigma_h)) = \text{act}(\alpha)$ |
| $\text{var}_e(\text{urgent}_{L_u}(\alpha)) = \text{var}_e(\alpha)$ | $\text{act}(\text{urgent}_{L_u}(\alpha)) = \text{act}(\alpha)$ |
| $\text{var}_e(\text{encap}_{L_e}(\alpha)) = \text{var}_e(\alpha)$ | $\text{act}(\text{encap}_{L_e}(\alpha)) = \text{act}(\alpha)$ |

**Definition 4.5** (Syntactic extensions)**.** The variable hiding operator $\text{hideact}_{L_h}(\alpha)$, the nondelayable action operator $\text{urgent}_{L_u}(\alpha)$ and the encapsulation operator $\text{encap}_{L_e}(\alpha)$, take as arguments possibly infinite sets $L_h$, $L_u$ and $L_e$, respectively, of actions. The sets can be infinite because an action that represents communication via a channel is composed of the channel label together with the communicated value. To avoid explicit notations of such infinite sets, three syntactic extensions are defined:

- $\text{hideact}_{L_h, H_h}(\alpha) \triangleq \text{hideact}_{L_h \cup \{h!cs, h?cs, h!?cs \mid h \in H_h, cs \in \Lambda^*\}}(\alpha)$,
- $\text{urgent}_{L_u, H_u}(\alpha) \triangleq \text{urgent}_{L_u \cup \{h!?cs \mid h \in H_h, cs \in \Lambda^*\}}(\alpha)$,
- $\text{encap}_{L_e, H_e}(\alpha) \triangleq \text{encap}_{L_e \cup \{h!cs, h?cs \mid h \in H_h, cs \in \Lambda^*\}}(\alpha)$,

where

- $L_h$, $L_u$, and $L_e$ are as defined in Definition 4.3
- $H_h \subseteq \mathcal{H}$ denotes a set of channels for which all CSP action labels (send, receive, synchronization and communication) are hidden;
- $H_u \subseteq \mathcal{H}$ denotes a set of channels for which all CSP synchronization and communication action labels are nondelayable;
- $H_e \subseteq \mathcal{H}$ denotes a set of channels for which the separate send and receive actions are blocked.

[1] This does not mean that these actions are actually used. It is allowed to specify the set of actions much broader than the actions that appear on transitions.

# 5 Semantics of interchange automata

The formal semantics associates to each interchange automaton an action transition relation, a time transition relation, and a consistency transition relation on states. A different way of looking at such a semantics is as a labeled transition system with three types of transitions. The states $S$ of the labeled transition system associated to an interchange automaton consist of an interchange automaton, a valuation of the external state variables of that automaton, and a set of jumping external state variables: i.e., $S = \mathcal{A} \times \mathit{Val} \times \mathcal{P}(\mathcal{V})$, where $\mathit{Val} = \mathcal{V} \cup \dot{\mathcal{V}} \mapsto \Lambda$ is the set of all partial mappings from $\mathcal{V} \cup \dot{\mathcal{V}}$ to the set of values $\Lambda$. The valuation $\sigma$ of a state $(\alpha, \sigma, J)$ of a transition system defines values for precisely the externally visible state variables, i.e., $\mathrm{dom}(\sigma) = \mathrm{var}_{\mathrm{e,st}}(\alpha)$ for all $(\alpha, \sigma, J) \in S$. For the set of jumping variables $J$, see the semantics of parallel composition in Section 5.2.

The intuition of an action transition $(\alpha, \sigma, J) \xrightarrow{\xi, \ell, W, \xi'} (\alpha', \sigma', J')$ is that the state $(\alpha, \sigma, J)$ executes a discrete action (with action label) $\ell$ with visible valuations $\xi, \xi'$, before and after execution of the action, respectively, and thereby transforms into the state $(\alpha', \sigma', J')$, where $\sigma'$ denotes the accompanying valuation of the automaton $\alpha'$, after the discrete action $\ell$ is executed. The set $W$ represents the external state variables that are allowed to change (jump) in this action transition. They need to be visible for synchronization in a parallel composition of interchange automata, see the semantics of parallel composition in Section 5.2.

The intuition of a time transition $(\alpha, \sigma, J) \overset{t, \rho}{\mapsto} (\alpha', \sigma', J')$ is that model time passes for $t$ time units, and the valuation at each time-point $s \in [0, t]$ is given by $\rho(s)$ for the externally visible variables. At the end-point $t$, the resulting state is $(\alpha', \sigma', J')$.

The intuition of the consistency transition $(\alpha, \sigma, J) \overset{\xi}{\rightsquigarrow} \alpha'$ is that the interchange automaton $\alpha$ is consistent with extended valuation $\xi$, which means that the initial conditions and invariants of all active locations of $\alpha$ are satisfied in $\xi$. After the consistency transition, the initial conditions of the resulting automaton $\alpha'$ define the initial values for the internal state (discrete and continuous) variables.

**Notation 5.1.** In this section, some notations and operators are used. These are defined as follows:

- $\restriction$ is the restriction operator on functions. If $f$ is a function, and $S$ is a set, $f \restriction S$ denotes the restriction of $f$ to $S$, that is, the function $g$ with $\mathrm{dom}(g) = \mathrm{dom}(f) \cap S$, such that $g(c) = f(c)$ for each $c \in \mathrm{dom}(g)$.
- $\downarrow$ is the projection operator on functions, which is used here on trajectories. For $\rho : T \mapsto (Y \to \Lambda)$, $S \subseteq Y$ and $x \in Y$, $\rho \downarrow S$ denotes the function $\rho' : T \mapsto (S \to \Lambda)$ such that $\rho'(t) = \rho(t) \restriction S$ for each $t \in T$; and $\rho \downarrow x$ denotes the function $f : T \mapsto \Lambda$ such that $f(t) = \rho(t)(x)$ for each $t \in \mathrm{dom}(\rho)$.
- For atomic interchange automaton $\alpha_{\mathrm{atom}}$ given by $(X, X_{\mathrm{i}}, \mathrm{dtype}, V, v, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$, $\alpha_{\mathrm{atom}}[v', \mathrm{init}'/v, \mathrm{init}]$ denotes the atomic interchange automaton obtained from atomic interchange automaton $\alpha_{\mathrm{atom}}$ by replacing $v$ by $v'$ and init by init′, and $\alpha_{\mathrm{atom}}[\mathrm{init}'/\mathrm{init}] = \alpha_{\mathrm{atom}}[v, \mathrm{init}'/v, \mathrm{init}]$.

## 5.1 Semantics of atomic interchange automata

**Definition 5.2** (Action transitions)**.** Consider an atomic interchange automaton $\alpha = (X, X_{\mathrm{i}}, \mathrm{dtype}, V, v, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$. The *action transition relation* $\_ \Rightarrow \_ \subseteq S \times (\mathit{Val} \times \mathcal{L}_\tau \times \mathcal{P}(\mathcal{V}) \times \mathit{Val}) \times S$ is for $(\alpha, \sigma, J), (\alpha', \sigma', J) \in S$, $\xi_{\mathrm{e}}, \xi'_{\mathrm{e}} \in \mathit{Val}$, $\ell \in \mathcal{L}_\tau$, and $W_{\mathrm{e}} \subseteq \mathcal{V}$, defined as follows: $(\alpha, \sigma, J) \xrightarrow{\xi_{\mathrm{e}}, \ell, W_{\mathrm{e}}, \xi'_{\mathrm{e}}} (\alpha', \sigma', J)$, if and only if there exist an edge $(v, g, a, (W, r), v') \in E$, valuations $\xi, \xi' \in \mathit{Val}$ with $\mathrm{dom}(\xi) = \mathrm{dom}(\xi') = \tilde{X}$, a list of values $[\mathbf{c}_n] \in \Lambda^*$, a channel $h \in \mathcal{H}$, expressions $\{\mathbf{e}_n\} \subseteq \mathrm{Expr}(\tilde{X})$, and variables $\{\mathbf{x}_n\} \subseteq \tilde{X}$ such that

- $\xi \restriction \tilde{X}_\mathrm{e} = \xi_\mathrm{e}$ and $\xi' \restriction \tilde{X}_\mathrm{e} = \xi'_\mathrm{e}$;
- $\xi_\mathrm{e} \restriction X_\mathrm{state} = \sigma$ and $\xi'_\mathrm{e} \restriction X_\mathrm{state} = \sigma'$;
- $\xi \models \mathrm{init}$ and $\xi \models g$;
- $\xi \models \mathrm{inv}(v)$ and $\xi' \models \mathrm{inv}(v')$;
- $\xi' \cup \xi^- \models r$;
- $\xi \restriction X_\mathrm{nonjmp} = \xi' \restriction X_\mathrm{nonjmp}$, where $X_\mathrm{nonjmp} = X_\mathrm{state} \setminus (W \cup (J \cap X_\mathrm{e}))$;
- $\begin{cases} \ell = a & \text{if } a \in L \\ \ell = h\,![\mathbf{c}_n], \text{ where } \mathbf{c}_n = \xi(\mathbf{e}_n), & \text{if } a = h\,!\,\mathbf{e}_n \\ \ell = h\,?[\mathbf{c}_n], \text{ where } \mathbf{c}_n = \xi'(\mathbf{x}_n) & \text{if } a = h\,?\,\mathbf{x}_n \\ \ell = h\,!?[\mathbf{c}_n], \text{ where } \mathbf{c}_n = \xi'(\mathbf{x}_n) = \xi(\mathbf{e}_n) & \text{if } a = h\,!?\,\mathbf{x}_n := \mathbf{e}_n \\ \ell = h\,!?[] & \text{if } a = h\,!? \end{cases}$
- $W_\mathrm{e} = W \cap X_\mathrm{state} \cap X_\mathrm{e}$;
- $\alpha' = \alpha[v', \mathrm{init}'/v, \mathrm{init}]$, $\mathrm{init}' = \left( \bigwedge_{x \in X_\mathrm{state} \cap X_\mathrm{i}} x = c_x \right)$, and $c_x \in \Lambda$ is given by $c_x = \xi'(x)$.

The notation *val* $\models$ *pred*, where *val* is a valuation and *pred* a predicate, means that *pred* is satisfied when all variables occurring in it are substituted by their values as defined in *val*. Minus superscripted variables, such as $x^-$, occurring in $r$ are evaluated in $\xi^-$, which is defined as $\mathrm{dom}(\xi^-) = \{x^- \mid x \in \mathrm{dom}(\xi)\}$, and $\xi^-(x^-) = \xi(x)$. The 'non-jumping' variables in the set $X_\mathrm{nonjmp} = X_\mathrm{state} \setminus (W \cup (J \cap X_\mathrm{e}))$ are the variables the values of which are not allowed to change in an action transition. These variables are the discrete and continuous variables apart from two sets of variables: the variables from set $W$ and the externally visible variables from set $J$ ($J \cap X_\mathrm{e}$). The jumping variables in set $J$ are the result of changes in external variables of synchronizing automata, as defined in the semantics of parallel composition Rules 1 and 2.

The notations $\mathbf{e}_n$, $\mathbf{c}_n$, $\mathbf{x}_n$, $\xi(\mathbf{e}_n)$, and $\xi'(\mathbf{x}_n)$ denote $e_1, \ldots, e_n$, $c_1, \ldots, c_n$, $x_1, \ldots, x_n$, $\xi(e_1), \ldots, \xi(e_n)$, and $\xi'(x_1), \ldots, \xi'(x_n)$, respectively, for $n \geq 1$. The values of expressions $e_1, \ldots, e_n$ which are sent via channel $h$ are evaluated in valuation $\xi$. The case that $n$ equals 0, represents the case where nothing is sent via the channel, and $\mathbf{e}_0$ and $[\mathbf{c}_0]$ denote an empty expression and an empty list [], respectively. For $n \geq 1$, the receive statement $h\,?\,x_1, \ldots, x_n$ can receive the list of values $[c_1, \ldots, c_n]$ (see also the semantics of the parallel composition operator in Section 5.2). For $n = 0$, nothing is received, so that $\mathbf{x}_0$ and $\mathbf{c}_0$ are empty, and $\xi'(\mathbf{x}_0) = \mathbf{c}_0$ always holds.

The updated initial condition $\mathrm{init}'$ acts as a local valuation which ensures that for each local state variable $x$, its starting value for the next transition equals its resulting value (here: $\xi'(x)$, in Definition 5.3: $\rho(t)(x)$) for the current transition. Note that we do not combine the valuation of the internal variables with the valuation of the external variables in $\sigma$. Having the valuations of the local variables in $\sigma$ would lead to restrictions on the parallel composition of two automata, namely that the sets of internal variables of two automata in a parallel composition would need to be disjoint. Otherwise, local variables with the same names in parallel automata would become shared. See for example [30], where the local variables and local actions of automata in a parallel composition are required to be disjoint.

**Definition 5.3** (Time transitions)**.** Consider an atomic interchange automaton $\alpha = (X, X_\mathrm{i}, \mathrm{dtype}, V, v, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$. The *time transition relation* $\_ \overset{\_}{\longmapsto} \_ \subseteq S \times (T \times (T \mapsto \mathit{Val})) \times S$ is for $(\alpha, \sigma, J), (\alpha', \sigma', J) \in S$, $t \in T$, and $\rho_\mathrm{e} : [0, t] \to \mathit{Val}$, defined as follows: $(\alpha, \sigma, J) \overset{t, \rho_\mathrm{e}}{\mapsto} (\alpha', \sigma', J)$, if and only if there exists a $\rho : [0, t] \to \mathit{Val}$ with $\mathrm{dom}(\rho(s)) = \tilde{X}$ for all $s \in [0, t]$ such that

- $\rho \downarrow \tilde{X}_\mathrm{e} = \rho_\mathrm{e}$;
- $\rho_\mathrm{e}(0) \restriction X_\mathrm{state} = \sigma$ and $\rho_\mathrm{e}(t) \restriction X_\mathrm{state} = \sigma'$;
- $\rho(0) \models \mathrm{init}$;
- $\rho \downarrow x$ is a constant function for all $x \in X_\mathrm{disc}$;
- $(\rho \downarrow x) \in F$ for all $x \in X_\mathrm{alg}$;
- $\rho \downarrow \dot{x}$ is an integrable function in the Lebesgue sense for all $x \in X_\mathrm{cont}$;
- $\rho(s) \models \mathrm{flow}(v)$ and $\rho(s) \models \mathrm{inv}(v)$ for all $s \in [0, t]$;
- $\rho(s) \models \mathrm{tcp}(v)$ for all $s \in \{0\} \cup [0, t)$;
- $(\rho \downarrow x)(s) = (\rho \downarrow x)(0) + \int_0^s (\rho \downarrow \dot{x})(s')ds'$ for all $x \in X_\mathrm{cont}$ and $s \in [0, t]$;
- $(\rho \downarrow x, \rho \downarrow \dot{x}) \in G$ for all $x \in X_\mathrm{cont}$;
- $\alpha' = \alpha[\mathrm{init}'/\mathrm{init}]$, $\mathrm{init}' = \left( \bigwedge_{x \in X_\mathrm{state} \cap X_\mathrm{i}} x = c_x \right)$, and $c_x \in \Lambda$ is given by $c_x = \rho(t)(x)$.

Item $(\rho \downarrow x) \in F$ for all $x \in X_\mathrm{alg}$, requires the trajectories of the algebraic variables to be functions of type $F$. This set of functions is a global parameter of the solution concept of an interchange automaton specification.

The trajectories of the dotted variables are required to be integrable. This ensures that the integral $\int_0^s (\rho \downarrow \dot{x})(s')ds'$ is defined. The relation between the trajectory of a continuous variable $x$ and the trajectory of its 'derivative' $\dot{x}$ is given by the Caratheodory solution concept [16]: $(\rho \downarrow x)(s) = (\rho \downarrow x)(0) + \int_0^s (\rho \downarrow \dot{x})(s')ds'$. This integral relation can hold only for those continuous variables for which $\rho \downarrow x$ is an absolutely continuous function, but it does allow a non-smooth trajectory for a continuous variable in the case that the trajectory of its 'derivative' $\rho \downarrow \dot{x}$ is non-smooth or even discontinuous, as in, for example, in the solution of the invariant $\dot{y} = \mathrm{step}(t - 1) \wedge \dot{t} = 1$, where $t$ and $y$ are continuous variable with initial value of 0, and $\mathrm{step}(x)$ equals 0 for $x \leq 0$ and 1 for $x > 0$.

In hybrid automata, the solution concept usually defines the function $\rho \downarrow \dot{x}$ to be the derivative function of $\rho \downarrow x$ for continuous variables $x \in X_\mathrm{cont}$. This can be realized for the interchange automaton format semantics by restricting the set $G$, which is used in the requirement $(\rho \downarrow x, \rho \downarrow \dot{x}) \in G$ for all $x \in X_\mathrm{cont}$, as $G = \{(f, f') \mid f \text{ is differentiable, and } f' \text{ is the derivative function of } f\}$. In this way, the semantics of the interchange automaton format corresponds to the usual semantics of hybrid automata. Piecewise continuous functions for the trajectories of the algebraic and dotted variables can be expressed by means of: $F = \{f \mid f \text{ is a piecewise continuous function}\}$, $G = \{(f, f') \mid f' \text{ is a piecewise continuous function}\}$. Another possibility would be not to define additional restrictions: $F = \{f \mid \mathrm{true}\}$, $G = \{(f, f') \mid \mathrm{true}\}$. For an invariant such as $\dot{x} = y,\ y = \mathrm{step}(t - 1),\ \dot{t} = 1$, which has just one solution for continuous variables $x, t$ and algebraic variable $y$, the solution is the same for both cases of $F$ and $G$. For an invariant 'true' that allows infinitely many solutions, there would obviously be a difference.

**Definition 5.4** (Consistency transitions)**.** Consider an atomic interchange automaton $\alpha = (X, X_\mathrm{i}, \mathrm{dtype}, V, v, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$. The *consistency transition relation* $\_ \overset{\sim}{\rightsquigarrow} \_ \subseteq S \times \mathit{Val} \times \mathcal{A}$ is for $(\alpha, \sigma, J) \in S, \alpha' \in \mathcal{A}$ and $\xi_\mathrm{e} \in \mathit{Val}$, defined as follows: $(\alpha, \sigma, J) \overset{\xi_\mathrm{e}}{\rightsquigarrow} \alpha'$, if and only if there exists a valuation $\xi \in \mathit{Val}$ with $\mathrm{dom}(\xi) = \tilde{X}$ such that

- $\xi \restriction \tilde{X}_\mathrm{e} = \xi_\mathrm{e}$;
- $\xi_\mathrm{e} \restriction X_\mathrm{state} = \sigma$;

- $\xi \models \text{init}$ and $\xi \models \text{inv}(v)$;
- $\alpha' = \alpha[\text{init}'/\text{init}]$, $\text{init}' = \left( \bigwedge_{x \in X_{\text{state}} \cap X_{\text{i}}} x = c_x \right)$, and $c_x \in \Lambda$ is given by $c_x = \xi(x)$.

Initially, the initial condition init may define conditions over all kinds of variables: internal or external, algebraic, continuous or discrete. After the first transition (action, time or consistency transition), however, the initial condition defines the values of the internal discrete and continuous variables only. The only case where the resulting automaton $\alpha'$ may differ from the original automaton $\alpha$ in a consistency transition is when the initial condition of the original automaton does not explicitly define the values for the internal discrete and continuous variables.

## 5.2 Semantics of the operators

The formal semantics of the operators is defined in a structured operational semantics (SOS) style [37] below.

**Notation 5.5.** The following notations are used in this section:

- For functions $f, g : A \mapsto B$, we define $f \doteq g$ iff $f(a) = g(a)$ for all $a \in \text{dom}(f) \cap \text{dom}(g)$. Similarly, for functions $f, g : T \mapsto A \mapsto B$ with $\text{dom}(f) = \text{dom}(g)$, we define $f \doteqdot g$ iff $f(t) \doteq g(t)$ for all $t \in \text{dom}(f)$.
- If $f$ and $g$ are functions with $f \doteq g$, then $f \cup g$ denotes the function $h$ with $\text{dom}(h) = \text{dom}(f) \cup \text{dom}(g)$ satisfying the condition: for each $c \in \text{dom}(h)$, if $c \in \text{dom}(f)$ then $h(c) = f(c)$, and $h(c) = g(c)$ otherwise. Similarly, if $f$ and $g$ are functions with $f \doteqdot g$, then $f \Cup g$ denotes the function $h$ with $\text{dom}(h) = \text{dom}(f)$ satisfying the condition: for each $t \in \text{dom}(h)$, $h(t) = f(t) \cup g(t)$.
- For functions $f, g : A \mapsto B$, we define $f \succ g$ to denote the function $h$ with $\text{dom}(h) = \text{dom}(f) \cup \text{dom}(g)$ satisfying the condition: for each $a \in \text{dom}(h)$, if $a \in \text{dom}(f)$ then $h(a) = f(a)$, and $h(a) = g(a)$ otherwise.
- The notation $\frac{H}{R}$, where $R$ is a number of results separated by commas, as used in Rules 2 and 3, is an abbreviation for a set of deduction rules of the form $\frac{H}{r}$; one for each $r \in R$.

### Parallel composition

There are no compatibility requirements for the parallel composition of interchange automata: any pair of interchange automata can be composed by the parallel composition operator. The parallel composition operator synchronizes on all basic actions that are shared in the alphabets of the arguments. CSP actions synchronize on the basis of pairs of matching send ($h!cs$) and receive ($h?cs$) actions, which are the result of send ($h!e$) and receive ($h?x$) statements on matching channels. All other actions may be interleaved (under the condition that they maintain the consistency of the other automaton). Time transitions must be synchronized, and consistency is established only if both automata agree on it. The external state variables that are shared by the argument automata need to have the same values (all the time).

$$\frac{\begin{array}{c}(\alpha_1, \sigma_{\alpha_1}, J \cup W_2) \xrightarrow{\xi_1, \ell, W_1, \xi_1'} (\alpha_1', \sigma_1', J'),\ (\alpha_2, \sigma_{\alpha_2}, J \cup W_1) \xrightarrow{\xi_2, \ell, W_2, \xi_2'} (\alpha_2', \sigma_2', J''), \\ \xi_1 \doteq \xi_2,\ \xi_1' \doteq \xi_2',\ l \in \text{act}(\alpha_1) \cap \text{act}(\alpha_2)\end{array}}{(\alpha_1 \parallel \alpha_2, \sigma, J) \xrightarrow{\xi_1 \cup \xi_2, \ell, W_1 \cup W_2, \xi_1' \cup \xi_2'} (\alpha_1' \parallel \alpha_2', \sigma_1' \cup \sigma_2', J)}\ 1$$

$$\frac{\begin{array}{c}(\alpha_1, \sigma_{\alpha_1}, J \cup W_2) \xrightarrow{\xi_1, h!cs, W_1, \xi_1'} (\alpha_1', \sigma_1', J'),\ (\alpha_2, \sigma_{\alpha_2}, J \cup W_1) \xrightarrow{\xi_2, h?cs, W_2, \xi_2'} (\alpha_2', \sigma_2', J''),\\ \xi_1 \doteq \xi_2,\ \xi_1' \doteq \xi_2'\end{array}}{\begin{array}{c}(\alpha_1 \parallel \alpha_2, \sigma, J) \xrightarrow{\xi_1 \cup \xi_2, h!?cs, W_1 \cup W_2, \xi_1' \cup \xi_2'} (\alpha_1' \parallel \alpha_2', \sigma_1' \cup \sigma_2', J)\\ (\alpha_2 \parallel \alpha_1, \sigma, J) \xrightarrow{\xi_2 \cup \xi_1, h!?cs, W_2 \cup W_1, \xi_2' \cup \xi_1'} (\alpha_2' \parallel \alpha_1', \sigma_1' \cup \sigma_2', J)\end{array}}\ 2$$

$$\frac{\begin{array}{c}(\alpha_2, \sigma_{\alpha_2}, J) \overset{\xi_2}{\rightsquigarrow} \alpha_2',\ (\alpha_1, \sigma_{\alpha_1}, J) \xrightarrow{\xi_1, \ell, W, \xi_1'} (\alpha_1', \sigma_{\alpha_1}', J''),\\ (\alpha_2', \sigma_{\alpha_1}' \restriction \mathrm{dom}(\sigma_{\alpha_2}) \succ \sigma_{\alpha_2}, J) \overset{\xi_2'}{\rightsquigarrow} \alpha_2'',\\ \xi_1 \doteq \xi_2,\ \xi_1' \doteq \xi_2',\ \ell \notin \mathrm{act}(\alpha_1) \cap \mathrm{act}(\alpha_2)\end{array}}{\begin{array}{c}(\alpha_1 \parallel \alpha_2, \sigma, J) \xrightarrow{\xi_1 \cup \xi_2, \ell, W, \xi_1' \cup \xi_2'} (\alpha_1' \parallel \alpha_2'', \sigma_{\alpha_1}' \succ \sigma, J),\\ (\alpha_2 \parallel \alpha_1, \sigma, J) \xrightarrow{\xi_2 \cup \xi_1, \ell, W, \xi_2' \cup \xi_1'} (\alpha_2'' \parallel \alpha_1', \sigma_{\alpha_1}' \succ \sigma, J)\end{array}}\ 3$$

$$\frac{(\alpha_1, \sigma_{\alpha_1}, J) \overset{t, \rho_1}{\longmapsto} (\alpha_1', \sigma_1', J'),\ (\alpha_2, \sigma_{\alpha_2}, J) \overset{t, \rho_2}{\longmapsto} (\alpha_2', \sigma_2', J''),\ \rho_1 \fatsemi \rho_2}{(\alpha_1 \parallel \alpha_2, \sigma, J) \overset{t, \rho_1 \Cup \rho_2}{\longmapsto} (\alpha_1' \parallel \alpha_2', \sigma_1' \cup \sigma_2', J)}\ 4$$

$$\frac{(\alpha_1, \sigma_{\alpha_1}, J) \overset{\xi_1}{\rightsquigarrow} \alpha_1',\ (\alpha_2, \sigma_{\alpha_2}, J) \overset{\xi_2}{\rightsquigarrow} \alpha_2',\ \xi_1 \doteq \xi_2}{(\alpha_1 \parallel \alpha_2, \sigma, J) \overset{\xi_1 \cup \xi_2}{\rightsquigarrow} \alpha_1' \parallel \alpha_2'}\ 5$$

Here, $\sigma_{\alpha_1}$ and $\sigma_{\alpha_2}$ are abbreviations for $\sigma \restriction \mathrm{var}_{\mathrm{e,st}}(\alpha_1)$ and $\sigma \restriction \mathrm{var}_{\mathrm{e,st}}(\alpha_2)$, respectively, and $\sigma_{\alpha_1}'$ denotes a (free) valuation (no abbreviation). Note that by definition (see the definition of the state $S$ in Section 5), $\mathrm{dom}(\sigma) = \mathrm{var}_{\mathrm{e,st}}(\alpha_1 \parallel \alpha_2)$. The valuation $\sigma_{\alpha_1}' \restriction \mathrm{dom}(\sigma_{\alpha_2}) \succ \sigma_{\alpha_2}$ in Rule 3 represents the updated version of valuation $\sigma_{\alpha_2}$ such that the valuations for the shared variables $(\mathrm{var}_{\mathrm{e,st}}(\alpha_1) \cap \mathrm{var}_{\mathrm{e,st}}(\alpha_2))$ in $\sigma_{\alpha_2}$ have been overwritten by the values defined in $\sigma_{\alpha_1}'$. In this way, Rule 3 allows an automaton to change the values of variables shared with another automaton, as long as the new valuations for the shared variables are consistent with the (initial conditions and invariants of the) other automaton.

In literature, ensuring that the invariants of non-synchronizing automata hold is usually accomplished in (one of) the following ways:

- The semantics of parallel composition is defined as a syntactic operation on the participating automata. The resulting automaton is defined as the Cartesian product of the participating automata, such that the invariant of each location of the resulting automaton is the conjunction of the invariants of the composed locations, see for example [1] and [3].
- The semantics of parallel composition is defined on automata that do not share variables. See for example [24], where "non-synchronizing automata" do in fact synchronize by means of a zero-delay time transition, which ensures that their local variables do not change.
- The semantics of parallel composition is defined as an operation on the transition systems of the participating automata using stutter transitions in [17], where the term environment transition is used instead of stutter transition. All "non-synchronizing" automata participate in each action transition by means of the stutter transition, which ensures that the invariants of the non-synchronizing automata hold.
- The semantics of parallel composition is defined on automata that have partitioned the variables into internal and external variables in the semantical hybrid I/O automata defined in [30]. The values of the internal variables of the non-synchronizing automata remain unchanged, and action transitions are defined only on the internal variables. The external variables are not part of the state, just like the algebraic variables in the CIF. The external variables are partitioned into input and output variables.

In [1] and [17], the stutter transition also enables partitioning of the variables into input and controlled variables. The input variables are allowed to change arbitrarily in a stutter transition, the controlled variables remain the same. Output variables, that are a subclass of the controlled variables, cannot be shared.

In the CIF, input and output variables can easily be defined in the concrete syntax as external variables. Input variables in the concrete syntax map to algebraic variables in the abstract syntax, and the values of input variables of an automaton should not be restricted by the automaton in action or time transitions, in precisely the same way as required for the input variables in the automata of [1] and [17]. Stutter transitions are not required in the CIF, because of the presence of algebraic variables. The semantics of parallel composition is defined as an operation on the transition systems of the participating automata using consistency transitions. The advantage of not partitioning the variables into inputs and controlled variables in the abstract format is that such a 'hard-coded' partitioning would prevent the behavioral approach [38] as used in languages such as Modelica [41] and EcosimPro [15], from being used. The CIF allows both the behavioral approach, by means of external variables, and input-output partitioning by allowing the declaration of inputs and outputs as a restricted kind of external variables in the concrete format as defined in Section 7.

### Variable hiding

The variable hiding operator applied to an automaton, $\text{hidevar}_{X_\text{h}}(\alpha, \sigma_\text{h})$, hides the variables from set $X_\text{h}$ by removing information about them from the action and time transitions of $\alpha$. The values of the hidden state variables are stored in valuation $\sigma_\text{h}$.

$$\frac{(\alpha, \sigma \cup \sigma'_\text{h}, J \setminus X_\text{h}) \xrightarrow{\xi, \ell, W, \xi'} (\alpha', \sigma', J'),\ \sigma'_\text{h} \doteq \sigma_\text{h},\ \text{dom}(\sigma'_\text{h}) = \text{var}_{\text{e,st}}(\alpha) \cap X_\text{h}}{(\text{hidevar}_{X_\text{h}}(\alpha, \sigma_\text{h}), \sigma, J) \xrightarrow{\xi_\text{vis}, \ell, W_\text{vis}, \xi'_\text{vis}} (\text{hidevar}_{X_\text{h}}(\alpha', \sigma'_{\sigma'_\text{h}}), \sigma'_\sigma, J)}\ 6$$

$$\frac{(\alpha, \sigma \cup \sigma'_\text{h}, J \setminus X_\text{h}) \overset{t, \rho}{\longmapsto} (\alpha', \sigma', J'),\ \sigma'_\text{h} \doteq \sigma_\text{h},\ \text{dom}(\sigma'_\text{h}) = \text{var}_{\text{e,st}}(\alpha) \cap X_\text{h}}{(\text{hidevar}_{X_\text{h}}(\alpha, \sigma_\text{h}), \sigma, J) \overset{t, \rho_\text{vis}}{\longmapsto} (\text{hidevar}_{X_\text{h}}(\alpha', \sigma'_{\sigma'_\text{h}}), \sigma'_\sigma, J)}\ 7$$

$$\frac{(\alpha, \sigma \cup \sigma'_\text{h}, J \setminus X_\text{h}) \overset{\xi}{\rightsquigarrow} \alpha',\ \sigma'_\text{h} \doteq \sigma_\text{h},\ \text{dom}(\sigma'_\text{h}) = \text{var}_{\text{e,st}}(\alpha) \cap X_\text{h}}{(\text{hidevar}_{X_\text{h}}(\alpha, \sigma_\text{h}), \sigma, J) \overset{\xi_\text{vis}}{\rightsquigarrow} \text{hidevar}_{X_\text{h}}(\alpha', \sigma_\text{h})}\ 8$$

Notations $\sigma'_{\sigma'_\text{h}}$ and $\sigma'_\sigma$ are abbreviations for $\sigma' \restriction \text{dom}(\sigma'_\text{h})$ and $\sigma' \restriction \text{dom}(\sigma)$, respectively. The visible valuations $\xi_\text{vis}$ and $\xi'_\text{vis}$, visible set of jumping variables $W_\text{vis}$, and visible trajectory $\rho_\text{vis}$, are abbreviations for $\xi \restriction (X_\text{vis} \cup \dot{X}_\text{vis})$, $\xi' \restriction (X_\text{vis} \cup \dot{X}_\text{vis})$, $W \restriction X_\text{vis}$, and $\rho \downarrow (X_\text{vis} \cup \dot{X}_\text{vis})$, respectively, with $X_\text{vis} = \text{var}_\text{e}(\alpha) \setminus X_\text{h}$.

Valuation $\sigma_\text{h} : X_\text{h} \mapsto \Lambda$ is a partial function that defines the initial values for (some of) the hidden variables. These initial values are relevant only for the hidden state variables: valuation $\sigma'_\text{h} : \text{var}_{\text{e,st}}(\alpha) \cap X_\text{h} \to \Lambda$ defines the initial values for the hidden state variables and takes them from $\sigma_\text{h}$ if possible and takes arbitrary values otherwise: $\sigma'_\text{h} \doteq \sigma_\text{h}$.

### Action hiding

The action hiding operator applied to an automaton, $\text{hideact}_{L_\text{h}}(\alpha)$, hides (abstracts from) the basic action labels from the set $L_\text{h}$ by replacing the actions to hide by the internal action $\tau$. This only affects the action behavior of $\alpha$; its delay behavior and consistency behavior remain unchanged.

$$\frac{(\alpha, \sigma, J) \xrightarrow{\xi, \ell, W, \xi'} (\alpha', \sigma', J'),\ \ell \notin L_\text{h}}{(\text{hideact}_{L_\text{h}}(\alpha), \sigma, J) \xrightarrow{\xi, \ell, W, \xi'} (\text{hideact}_{L_\text{h}}(\alpha'), \sigma', J)}\ 9$$

$$\frac{(\alpha, \sigma, J) \xrightarrow{\xi,\ell,W,\xi'} (\alpha', \sigma', J'),\ \ell \in L_{\mathrm{h}}}{(\mathrm{hideact}_{L_{\mathrm{h}}}(\alpha), \sigma, J) \xrightarrow{\xi,\tau,W,\xi'} (\mathrm{hideact}_{L_{\mathrm{h}}}(\alpha'), \sigma', J)}\ 10$$

$$\frac{(\alpha, \sigma, J) \overset{t,\rho}{\longmapsto} (\alpha', \sigma', J')}{(\mathrm{hideact}_{L_{\mathrm{h}}}(\alpha), \sigma, J) \overset{t,\rho}{\longmapsto} (\mathrm{hideact}_{L_{\mathrm{h}}}(\alpha'), \sigma', J)}\ 11$$

$$\frac{(\alpha, \sigma, J) \overset{\xi}{\rightsquigarrow} \alpha'}{(\mathrm{hideact}_{L_{\mathrm{h}}}(\alpha), \sigma, J) \overset{\xi}{\rightsquigarrow} \mathrm{hideact}_{L_{\mathrm{h}}}(\alpha')}\ 12,$$

### Urgent action operator

The urgent action operator applied to an automaton, $\mathrm{urgent}_{L_{\mathrm{u}}}(\alpha)$, gives actions from the set $L_{\mathrm{u}}$. The action behavior and consistency behavior of $\alpha$ are not affected by the urgent action operator. Time transitions are allowed only if at the current state, and at each intermediate state while delaying, no actions from set $L_{\mathrm{u}}$ are possible.

$$\frac{(\alpha, \sigma, J) \xrightarrow{\xi,\ell,W,\xi'} (\alpha', \sigma', J')}{(\mathrm{urgent}_{L_{\mathrm{u}}}(\alpha), \sigma, J) \xrightarrow{\xi,\ell,W,\xi'} (\mathrm{urgent}_{L_{\mathrm{u}}}(\alpha'), \sigma', J)}\ 13$$

$$\frac{(\alpha, \sigma, J) \overset{\xi}{\rightsquigarrow} \alpha'}{(\mathrm{urgent}_{L_{\mathrm{u}}}(\alpha) \overset{\xi}{\rightsquigarrow} \mathrm{urgent}_{L_{\mathrm{u}}}(\alpha')}\ 14$$

$$\frac{\begin{array}{r}(\alpha, \sigma, J) \overset{t,\rho}{\longmapsto} (\alpha', \sigma', J'),\ \forall_{\ell \in L_{\mathrm{u}}} (\alpha, \sigma, J) \overset{\ell}{\not\rightarrow},\\ \forall_{s \in [0,t)}\ (\exists_{\alpha'', \sigma'', J''}\quad (\alpha, \sigma, J) \overset{s,\rho\restriction[0,s]}{\longmapsto} (\alpha'', \sigma'', J''),\\ (\alpha'', \sigma'', J'') \overset{t-s,\rho_{-s}}{\longmapsto} (\alpha', \sigma', J'),\ \forall_{\ell \in L_{\mathrm{u}}} (\alpha'', \sigma'', J'') \overset{\ell}{\not\rightarrow})\end{array}}{(\mathrm{urgent}_{L_{\mathrm{u}}}(\alpha), \sigma, J) \overset{t,\rho}{\longmapsto} (\mathrm{urgent}_{L_{\mathrm{u}}}(\alpha'), \sigma', J)}\ 15$$

Here $\rho_{-s}$ denotes the trajectory $\rho$ shifted left by $s$ time-units and starting at 0: $\mathrm{dom}(\rho_{-s}) = [0, t-s]$, assuming $\mathrm{dom}(\rho) = [0, t]$, and $\forall_{t' \in \mathrm{dom}(\rho_{-s})}\ \rho_{-s}(t') = \rho(t'+s)$. Furthermore, notation $(\alpha, \sigma, J) \overset{\ell}{\not\rightarrow}$ denotes $\neg\exists_{\xi, W, \xi', \alpha', \sigma', J'}\ (\alpha, \sigma, J) \xrightarrow{\xi,\ell,W,\xi'} (\alpha', \sigma', J')$.

### Action encapsulation operator

The action encapsulation operator applied to an automaton, $\mathrm{encap}_{L_{\mathrm{e}}}(\alpha)$, blocks actions from the set $L_{\mathrm{e}}$. The delay behavior and consistency behavior of $\alpha$ are not affected.

Send and receive actions on channels from a set $H_{\mathrm{e}}$ can be blocked by means of $\mathrm{encap}_{\{h!cs, h?cs \mid h \in H_{\mathrm{e}}, cs \in \Lambda^*\}}(\alpha)$, which can be abbreviated as $\mathrm{encap}_{\emptyset, H_{\mathrm{e}}}(\alpha)$, see Definition 4.5. In this way, only the the synchronous execution of matching send and receive actions via channels from the set $H_{\mathrm{e}}$ can take place.

$$\frac{(\alpha, \sigma, J) \xrightarrow{\xi,\ell,W,\xi'} (\alpha', \sigma', J'),\ \ell \notin L_{\mathrm{e}}}{(\mathrm{encap}_{L_{\mathrm{e}}}(\alpha), \sigma, J) \xrightarrow{\xi,\ell,W,\xi'} (\mathrm{encap}_{L_{\mathrm{e}}}(\alpha), \sigma', J)}\ 16$$

$$\frac{(\alpha, \sigma, J) \overset{t,\rho}{\longmapsto} (\alpha', \sigma', J')}{(\mathrm{encap}_{L_{\mathrm{e}}}(\alpha), \sigma, J) \overset{t,\rho}{\longmapsto} (\mathrm{encap}_{L_{\mathrm{e}}}(\alpha'), \sigma', J)}\ 17$$

$$\frac{(\alpha, \sigma, J) \overset{\xi}{\rightsquigarrow} \alpha'}{(\mathrm{encap}_{L_e}(\alpha), \sigma, J) \overset{\xi}{\rightsquigarrow} \mathrm{encap}_{L_e}(\alpha')} \; 18$$

### 5.3 Equality and compositionality

Two interchange automata $\alpha_1$ and $\alpha_2$ are considered *comparable* iff they have the same sets of externally visible state variables and actions: i.e., $\mathrm{var}_{e,st}(\alpha_1) = \mathrm{var}_{e,st}(\alpha_2)$ and $\mathrm{act}(\alpha_1) = \mathrm{act}(\alpha_2)$.

The externally visible state variables need to be the same, because the valuation $\sigma$ of a state $(\alpha, \sigma, J)$ of a transition system defines values for precisely the externally visible state variables, i.e., $\mathrm{dom}(\sigma) = \mathrm{var}_{e,st}(\alpha)$ for all $(\alpha, \sigma, J) \in S$.

The externally visible actions need to be the same for the following reason. Consider two automata that have different externally visible actions but that are otherwise identical. These automata have the same transition systems, and can thus not be distinguished by means of bisimulation, but they may behave differently in a parallel composition with a third automaton.

Next, we define when two comparable interchange automata are equivalent. For this we use the notion of stateless bisimilarity as defined in [34].

**Definition 5.6.** A symmetric relation $R$ on comparable interchange automata is a (stateless) bisimulation relation, if and only if for all $(\alpha_1, \alpha_2) \in R$ and for all $\alpha_1', \sigma, \sigma', J, J', \xi, \xi', \ell, W, t, \rho$:

- if $(\alpha_1, \sigma, J) \xrightarrow{\xi, \ell, W, \xi'} (\alpha_1', \sigma', J')$, then $(\alpha_2, \sigma, J) \xrightarrow{\xi, \ell, W, \xi'} (\alpha_2', \sigma', J')$ for some $\alpha_2'$ such that $(\alpha_1', \alpha_2') \in R$;
- if $(\alpha_1, \sigma, J) \overset{t,\rho}{\longmapsto} (\alpha_1', \sigma', J')$, then $(\alpha_2, \sigma, J) \overset{t,\rho}{\longmapsto} (\alpha_2', \sigma', J')$ for some $\alpha_2'$ such that $(\alpha_1', \alpha_2') \in R$;
- if $(\alpha_1, \sigma, J) \overset{\xi}{\rightsquigarrow} \alpha_1'$, then $(\alpha_2, \sigma, J) \overset{\xi}{\rightsquigarrow} \alpha_2'$ for some $\alpha_2'$ such that $(\alpha_1', \alpha_2') \in R$.

Two interchange automata $\alpha_1$ and $\alpha_2$ are (stateless) *bisimilar*, notation $\alpha_1 \underline{\leftrightarrow} \alpha_2$, if and only if there exists a (stateless) bisimulation relation $R$ such that $(\alpha_1, \alpha_2) \in R$.

Note that if an automaton $\alpha_1$ is comparable with another automaton $\alpha_2$, then any automaton $\alpha_1'$ that can be reached from automaton $\alpha_1$ by performing action and time transitions is also comparable with $\alpha_2$. Therefore, the above restriction on a bisimulation relation that only comparable automata can be related needs to be verified/established only for the automata that one wishes to compare.

**Theorem 5.7.** *Bisimilarity is a congruence for all operators introduced in this article.*

*Proof.* The rules for action hiding and the action encapsulation operator satisfy the *process-tyft* format containing predicates [34] (which we call process-path format for simplicity). Thus for these operators, it can be concluded that stateless bisimilarity is a congruence.

The rules for parallel composition and variable hiding satisfy the process-tyft format except for a side-condition that refers to an automaton. However, since the truth value of these side-conditions is invariant with respect to comparable automata, (and any two stateless bisimilar automata are comparable) also for these it can be concluded that stateless bisimilarity is a congruence.

For the urgent action operator, Rule 15 does not satisfy the process-path format. Therefore we need to give manual proof to show that stateless bisimilarity is a congruence for the urgent action operator. The proof required is similar to the proof given in [31], pages 160–161. From this proof, we can conclude that stateless bisimilarity is also a congruence for the urgent action operator. □

# 6 Elimination of the operators

This section defines how an interchange automaton specification can be rewritten as a single atomic interchange automaton that is bisimilar to the original specification. For this purpose, we define how the operators of the interchange automaton language can be eliminated when they are applied to one or more atomic interchange automata.

## 6.1 Elimination of parallel composition

Let $\alpha_1$ and $\alpha_2$ denote two atomic interchange automata the shared variables of which have compatible types (defined below), and let two renaming functions $R_1$ and $R_2$ be defined on atomic interchange automata, in such a way, that the internal variables of the renamed automata $R_1(\alpha_1)$ and $R_2(\alpha_2)$ become unique (different from all other variables). The parallel composition operator can then be eliminated as defined below.

Let $R_1(\alpha_1) = (X_1,\ X_{\mathrm{i}1},\ \mathrm{dtype}_1,\ V_1,\ v_{0_1},\ \mathrm{init}_1,\ \mathrm{flow}_1,\ \mathrm{inv}_1,\ \mathrm{tcp}_1,\ L_1,\ E_1)$ and $R_2(\alpha_2) = (X_2,\ X_{\mathrm{i}2}, \mathrm{dtype}_2,\ V_2,\ v_{0_2}, \mathrm{init}_2, \mathrm{flow}_2, \mathrm{inv}_2, \mathrm{tcp}_2, L_2, E_2)$ denote the two atomic interchange automata that are obtained by renaming the internal variables of $\alpha_1$ and $\alpha_2$, so that $X_{\mathrm{i}1} \cap X_2 = \emptyset$ and $X_{\mathrm{i}2} \cap X_1 = \emptyset$. Then the parallel composition $\alpha_1 \parallel \alpha_2$ can be rewritten as an atomic interchange automaton $\alpha = (X,\ X_{\mathrm{i}},\ \mathrm{dtype},\ V,\ v_0,\ \mathrm{init},\ \mathrm{flow},\ \mathrm{inv},\ \mathrm{tcp},\ L,\ E)$ such that:

- $X = X_1 \cup X_2$;
- $X_{\mathrm{i}} = X_{\mathrm{i}1} \cup X_{\mathrm{i}2}$;
- for all $x \in X$: $\mathrm{dtype}(x) = \begin{cases} \mathrm{dtype}_1(x) & \text{if } x \in X_1 \setminus X_2 \\ \mathrm{dtype}_2(x) & \text{if } x \in X_2 \setminus X_1 \\ \mathrm{mdt}(\mathrm{dtype}_1(x), \mathrm{dtype}_2(x)) & \text{if } x \in X_1 \cap X_2 \end{cases}$
- $V = V_1 \times V_2$;
- $v_0 = (v_{0_1}, v_{0_2})$;
- $\mathrm{init} = \mathrm{init}_1 \wedge \mathrm{init}_2$;
- $\mathrm{flow}(v_1, v_2) = \mathrm{flow}_1(v_1) \wedge \mathrm{flow}_2(v_2)$, $\mathrm{inv}(v_1, v_2) = \mathrm{inv}_1(v_1) \wedge \mathrm{inv}_2(v_2)$, $\mathrm{tcp}(v_1, v_2) = \mathrm{tcp}_1(v_1) \vee \mathrm{tcp}_2(v_2)$;
- $L = L_1 \cup L_2$;
- $E$ contains an edge $((v_1, v_2), g, a, (W, r), (v_1', v_2'))$ if:
  - there is an edge $(v_2, g, a, (W, r), v_2') \in E_2$ such that $a \notin L_1$, and $v_1 = v_1'$, or
  - there is an edge $(v_1, g, a, (W, r), v_1') \in E_1$ such that $a \notin L_2$, and $v_2 = v_2'$.

  Note that in these two cases, $a$ may be a basic action label ($a \in \mathcal{L}_{\mathrm{basic}}$) or a CSP statement (send statement $h!\mathbf{e}$, receive statement $h?\mathbf{x}$, or communication/synchronization statement $h\,!?\,\mathbf{x} := \mathbf{e}/\ h!?$). CSP statements by definition cannot be in sets $L_1$ or $L_2$, since $\mathcal{L}_{\mathrm{basic}} \cap C_X = \emptyset$ (see the set of edges $E$ in Definition 4.2).
- $E$ contains the edge $((v_1, v_2), g_1 \wedge g_2, a, (W_1 \cup W_2, r_1 \wedge r_2), (v_1', v_2'))$ if there is an edge $(v_1, g_1, a, (W_1, r_1), v_1') \in E_1$ and an edge $(v_2, g_2, a, (W_2, r_2), v_2') \in E_2$ such that $a \in L_1 \cap L_2$. In this case, $a$ is a common basic action label ($a \in \mathcal{L}_{\mathrm{basic}}$).
- $E$ contains the edge $((v_1, v_2), g_1 \wedge g_2, h\,!?\,\mathbf{x} := \mathbf{e}, (W_1 \cup W_2, r_1 \wedge r_2), (v_1', v_2'))$ if:
  - there is an edge $(v_1, g_1, h!\mathbf{e}, (W_1, r_1), v_1') \in E_1$ and an edge $(v_2, g_2, h?\mathbf{x}, (W_2, r_2), v_2') \in E_2$, or

- there is an edge $(v_1,\ g_1,\ h?\mathbf{x},\ (W_1,\ r_1),\ v_1') \in E_1$ and an edge $(v_2, g_2, h!\mathbf{e}, (W_2, r_2), v_2') \in E_2$.

- $E$ contains the edge $((v_1, v_2), g_1 \wedge g_2, h\ !?, (W_1 \cup W_2, r_1 \wedge r_2), (v_1', v_2'))$ if:
  - there is an edge $(v_1, g_1, h!, (W_1, r_1), v_1') \in E_1$ and an edge $(v_2, g_2, h?, (W_2, r_2), v_2') \in E_2$, or
  - there is an edge $(v_1, g_1, h?, (W_1, r_1), v_1') \in E_1$ and an edge $(v_2, g_2, h!, (W_2, r_2), v_2') \in E_2$.

The function $\mathrm{mdt} \in \mathcal{D} \to \mathcal{D}$ merges the dynamic types from set $\mathcal{D} = \{\mathsf{disc}, \mathsf{cont}, \mathsf{alg}\}$ as follows: $\mathrm{mdt}(\mathsf{disc}, \mathsf{disc}) = \mathsf{disc}$, $\mathrm{mdt}(\mathsf{cont}, \mathsf{cont}) = \mathsf{cont}$, and $\mathrm{mdt}(\mathsf{alg}, d) = \mathrm{mdt}(d, \mathsf{alg}) = d$ for all $d \in \mathcal{D}$. Note that it is not correct to define $\mathrm{mdt}(\mathsf{cont}, \mathsf{disc}) = \mathrm{mdt}(\mathsf{disc}, \mathsf{cont}) = \mathsf{cont}$ and to define an additional equation $\dot{x} = 0$ for all variables that are of dynamic type disc in one of the parallel automata and of dynamic type cont in the other automaton. The restriction $\dot{x} = 0$ would be too strong, because of the Caratheodory solution concept (see time transitions in Section 5.1), which allows discontinuities (so unequal to zero) in the trajectory of $\dot{x}$ to obtain a constant trajectory for $x$. Therefore, compatible types are defined as follows: the dynamic type alg is compatible with all other dynamic types and identical types are compatible. Note that the requirement of type compatibility is required only for elimination of the parallel composition operator. The semantics of parallel automata is defined for all cases, also for the case that shared variables have incompatible types. We conjecture that the automata $\alpha_1 \parallel \alpha_2$ and $\alpha$ are bisimilar.

Renaming the internal variables is needed in general, because elimination of the parallel composition operator merges the variables of the two parallel automata, and only the external variables with the same names should be shared. This renaming of internal variables does not change the meaning of the automata, because the internal variables do not appear on the transition system (see Section 5). Note that elimination of the parallel composition operator does not require actions to be renamed, because the actions of sets $L$, $L_1$, $L_2$ are all external. Hidden actions are renamed to the non-synchronizing $\tau$ action label, either by the action hiding operator, or by elimination (see Sections 5.2 and 6.3, respectively).

### 6.2 Elimination of variable hiding

Let $\alpha = (X, X_\mathrm{i}, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$ denote an atomic interchange automaton. The variable hiding operator applied to this atomic interchange automaton $\mathrm{hidevar}_{X_\mathrm{h}}(\alpha, \sigma_\mathrm{h})$ can be rewritten as an atomic interchange automaton $\alpha' = (X, X_\mathrm{i}', \mathrm{dtype}, V, v_0, \mathrm{init}', \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$ such that:

- $X_\mathrm{i}' = X_\mathrm{i} \cup (X_\mathrm{h} \cap X_\mathrm{e})$;
- $\mathrm{init}' = \mathrm{init} \wedge \left( \bigwedge_{x \in \mathrm{dom}(\sigma_\mathrm{h}) \cap X_\mathrm{e} \cap X_\mathrm{state}} x = c_x \right)$, where $c_x \in \Lambda$ is given by $c_x = \sigma_\mathrm{h}(x)$.

We conjecture that the automata $\mathrm{hidevar}_{X_\mathrm{h}}(\alpha, \sigma_\mathrm{h})$ and $\alpha'$ are bisimilar.

### 6.3 Elimination of action hiding

Let $\alpha = (X, X_\mathrm{i}, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$ denote an atomic interchange automaton. The action hiding operator applied to this atomic interchange automaton $\mathrm{hideact}_{L_\mathrm{h}, H_\mathrm{h}}(\alpha)$ can be rewritten as an atomic interchange automaton $\alpha' = (X, X_\mathrm{i}, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L', E')$ such that:

- $L' = L \setminus L_\mathrm{h}$;
- $E' = \{(v, g, a, (W, r), v') \mid (v, g, a, (W, r), v') \in E, a \notin L_\mathrm{h} \cup C_{H_\mathrm{h}}\} \cup$
  $\{(v, g, \tau, (W, r), v') \mid (v, g, a, (W, r), v') \in E, a \in L_\mathrm{h} \cup C_{H_\mathrm{h}}\}$,

where $C_{H_h} = \{\ h!\mathbf{e}, h?\mathbf{x}, h!?, h!?\mathbf{x} := \mathbf{e} \mid h \in H_h, \{\mathbf{e}\} \subseteq \mathrm{Expr}(\tilde{X}), \{\mathbf{x}\} \subseteq \tilde{X}\ \}$. We conjecture that the automata $\mathrm{hideact}_{L_h, H_h}(\alpha)$ and $\alpha'$ are bisimilar.

### 6.4 Elimination of the action encapsulation operator

Let $\alpha = (X, X_i, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$ denote an atomic interchange automaton. The action encapsulation operator applied to this atomic interchange automaton $\mathrm{encap}_{L_e, H_e}(\alpha)$ can be rewritten as an atomic interchange automaton $\alpha' = (X, X_i, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E')$ such that:

- $E' = \{\ (v, g, a, (W, r), v')$
  $\mid (v, g, a, (W, r), v') \in E,\ a \notin \{h!\mathbf{e}, h?\mathbf{x} \mid h \in H_e, \{\mathbf{e}\} \subseteq \mathrm{Expr}(\tilde{X}), \{\mathbf{x}\} \subseteq \tilde{X}\}$
  $\}$

We conjecture that the automata $\mathrm{encap}_{L_e, H_e}(\alpha)$ and $\alpha'$ are bisimilar.

### 6.5 Elimination of the urgent action operator

For a subclass of atomic interchange automata characterized as being 'reactive' (defined below) for all urgent actions, the urgent action operator can be eliminated by redefining the tcp predicate. Let $\alpha = (X, X_i, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$ denote an atomic interchange automaton, let $L_u$ be a set of urgent actions and $H_u$ be a set of urgent channels, and let automaton $\alpha$ be reactive for all actions from $L_u$ and channels from $H_u$. Then the result of applying the urgent action operator with urgent action set $L_u$ and urgent channel set $H_u$ on $\alpha$, denoted as $\mathrm{urgent}_{L_u, H_u}(\alpha)$, can be rewritten as the interchange automaton $\alpha' = (X, X_i, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}', L, E)$, such that the tcp predicate of each location in $\alpha'$ is the conjunction of 1) the tcp predicate of the location in $\alpha$, and 2) the negated guards of the outgoing urgent edges of the location, where an urgent edge is an edge with either an action from $L_u$ or a CSP synchronization or communication statement via a channel from $H_u$:

$$\mathrm{tcp}'(v) = \mathrm{tcp}(v)\ \vee \left( \bigvee_{g \in \{g \mid (v, g, a, (W, r), v') \in E_u(v)\}} g \right) \text{ for all } v \in V,$$

where the set $E_u(v)$ of urgent edges at location $v$ is defined as $E_u(v) = \{(v, g, a, (W, r), v') \in E \mid a \in L_u \cup C_{H_u}\}$, and $C_{H_u} = \{\ h!?, h!?\mathbf{x} := \mathbf{e} \mid h \in H_u, \{\mathbf{e}\} \subseteq \mathrm{Expr}(\tilde{X}), \{\mathbf{x}\} \subseteq \tilde{X}\ \}$. We conjecture that the automata $\mathrm{urgent}_{L_u, H_u}(\alpha)$ and $\alpha'$ are bisimilar.

An atomic interchange automaton is reactive for a set of (urgent) actions $L_u$ and a set of (urgent) channels $H_u$ if whenever the guard of an urgent edge can be satisfied, the automaton has an action transition for an action from $L_u$, or for a CSP action via a channel from $H_u$. Here, an urgent edge is defined as an edge labeled with an urgent action $a \in L_u$ or labeled with a CSP synchronization or communication statement via a channel from $H_u$. This means that the invariants of the target locations of edges with urgent actions (and the jump predicates of the edges) should not prevent the action from taking place. A necessary condition for reactiveness would then be that from any reachable state, whenever the guard of an urgent edge can be satisfied, the automaton has an action transition for an action from $L_u$, or for a CSP action via a channel from $H_u$. A sufficient, but not necessary, condition for reactiveness is that from any state that satisfies the invariant of the location, whenever the guard of an urgent edge of that location can be satisfied, the automaton has an action transition for an action from $L_u$, or for a CSP action via a channel from $H_u$. A complicating factor is that the initial condition of a location may change after a transition, because the valuation for the local variables is stored in the initial condition of the active / initial location. The valuation $\sigma$ of a state $(\alpha, \sigma, J)$ only defines the values of the externally visible variables. Therefore, the sufficient condition for reactiveness is specified for arbitrary initial conditions. Formally, this sufficient condition can be defined as follows.

An atomic interchange automaton $(X, X_i, \mathrm{dtype}, V, v_0, \mathrm{init}, \mathrm{flow}, \mathrm{inv}, \mathrm{tcp}, L, E)$ is reactive for a

set of urgent actions $L_u \subseteq L$ and a set of urgent channels $H_u$ if

$$\forall_{(v,\xi,\text{init}',J)\in\hat{S}}\ (\forall_{(v,g,a,(W,r),v')\in E_u(v)}\ \xi \models g \implies (\alpha_{(v,\text{init}')}, \xi \restriction X_{\text{state}}, J) \xrightarrow{\xi_e,\ell_u,W,\xi_e'} (\alpha', \sigma, J)),$$

for some $\xi_e$, $W$, $\xi_e'$, $\alpha'$, and $\sigma$. Here the set of admissible extended states $\hat{S}$ is defined as: $\hat{S} = \{(v, \xi, \text{init}', J) \in V \times Val \times \text{Pred}(\tilde{X}) \times \mathcal{P}(\mathcal{V}) \mid \xi \models \text{inv}(v) \wedge \text{init}'\}$, the set of urgent edges $E_u(v)$ is defined above, $\alpha_{(v,\text{init}')}$ is defined as the atomic interchange automaton $\alpha$ with initial location $v$ and initial condition init$'$: $\alpha_{v,\text{init}'} = (X, X_i, \text{dtype}, V, v, \text{init}', \text{flow}, \text{inv}, \text{tcp}, L, E)$, and $\ell_u \in L_u \cup \{h!?cs \mid h \in H_u, cs \in \Lambda^*\}$.

# 7 Concrete syntax definition

In this section, the concrete syntax of CIF models is defined using a Backes-Naur (BNF) like notation. The symbol | defines choice, and notation [$Z$] defines $Z$ as being optional.

*spec* ::= [*defaultDefs*] [*autDefs*] *model* [*autDefs*]
*defaultDefs* ::= defaults *defaults*
*defaults* ::= *default* | *defaults* , *default*
*default* ::= mode *modeDefaults*
| edge *edgeDefaults*
| autoconnect
*modeDefaults* ::= *modeDefault* | *modeDefaults* , *modeDefault*
*modeDefault* ::= flow (true | false)
| tcp (true | false)
*edgeDefaults* ::= *edgeDefault* | *edgeDefaults* , *edgeDefault*
*edgeDefault* ::= (urgent | nonurg)
| (jump | nonjump)
*model* ::= model modelId [‘(’ *paramDecls* ‘)’] = *closedScope*
*autDefs* ::= *autDef* | *autDefs autDef*
*autDef* ::= automaton autId [‘(’ *paramDecls* ‘)’] = *closedScope*
*paramDecls* ::= *paramDecl* | *paramDecls*, *paramDecl*
*paramDecl* ::= *varIds* : type
*varIds* ::= varId | *varIds*, varId
*automaton* ::= *atomicAut*
| [autId:] *closedScope*
| [autId:] *openScope*
| [autId:] autId [ ‘(’ *varIdExprs* ‘)’ ]
| *automaton* ‖ *automaton*
| urgent(*actIds* , *chanIds* :: *automaton*)
| encap(*chanIds* :: *automaton*)
*atomicAut* ::= |( [*init*, ] *modes* :: modeId )|
*init* ::= init *preds*
*preds* ::= pred | *preds*, pred
*modes* ::= *mode* | *modes*, *mode*
*mode* ::= mode modeId = [*dyns*] [*edges*]
*dyns* ::= *dyn* | *dyns dyn*
*dyn* ::= (inv | flow | tcp) *preds*
*edges* ::= *edge* | *edges edge*
*edge* ::= [urgent | nonurg] [*guard*] [*action*] [*update*]
to modeId
*guard* ::= when *preds*
*action* ::= act (actId | *comLabel*)

*comLabel* ::= chanId ! [*exprs*]
| chanId ? [*varIds*]
| chanId !? [*varIds* := *exprs*]
*exprs* ::= expr | *exprs*, expr
*update* ::= do *varClockIds* := *exprs* | do *varClockIds* : *updpreds*
*varClockIds* ::= *varClockId*
| *varClockIds*, *varClockId*
*varClockId* ::= *varId*
| *clockId*
*updpreds* ::= updpred | *updpreds* & updpred
*closedScope* ::= ‖[ [*scopeDecls* ::] *automaton* ]‖
*openScope* ::= |( [*scopeDecls* ::] *automaton* )|
*scopeDecls* ::= *scopeDecl* | *scopeDecls*, *scopeDecl*
*scopeDecl* ::= extern *decls*
| input var *varIds* : [alg] type
| output var *varDecls*
| intern *decls*
| connect *connectSets*
*decls* ::= *decl* | *decls*, *decl*
*decl* ::= var *varDecls*
| clock *clockIds*
| chan (urgent | nonurg) *chanDecls*
| actlabel (urgent | nonurg) *actIds*
*varDecls* ::= *varDecl* | *varDecls*, *varDecl*
*varDecl* ::= *varIds* : (disc | cont | alg) type
[= (*expr* | '(' *exprs* ')')]
*clockIds* ::= clockId | *clockIds*, clockId
*chanDecls* ::= *chanDecl* | *chanDecls*, *chanDecl*
*chanDecl* ::= *chanIds* [! | ?] : type
*chanIds* ::= chanId | *chanIds*, chanId
*actIds* ::= actId | *actIds*, actId
*connectSets* ::= *connectSet* | *connectSets*, *connectSet*
*connectSet* ::= { *connector*} | {*connectors*, *connector* }
*connector* ::= [autId.] (varId | clockId | chanId | actId)
*varIdExprs* ::= *varIdExpr* | *varIdExprs*, *varIdExpr*
*varIdExpr* ::= varId = *expr*

Here, modelId denotes a model identifier, autId denotes an automaton identifier, varId denotes a variable identifier, clockId denotes a clock identifier, chanId denotes a channel identifier, actId denotes an action, modeId denotes a mode identifier, pred denotes a predicate over variables and dotted variables, updpred denotes a predicate over variables and primed variables, expr denotes an expression, and type denotes a static type.

## 7.1 Closed and open scopes

The building blocks that support hierarchy and modularity are the closed scope ‖[ *scopeDecls* :: *automaton* ]‖ and open scope |( *scopeDecls* :: *automaton* )|. Here, the double colon acts as a separator between the declarations of variables, action labels and channels on the one hand, and the specification of the behavior on the other hand. Declarations can be internal (intern keyword) or external (extern, input or output keyword). The main difference between closed and open scopes is that open scopes may have *free* variables, that is variables that are not bound in the open scope itself, but in an outer, more global, encompassing scope. A closed scope, on the other hand, may not contain free variables: all variables must be bound in the closed scope.

Both concepts of scoping can be found in modeling languages. The concept of open scopes is often used in different syntactic forms, such as the "for loop" used in many programming and modeling languages. It uses a local iterator that can be used in the body of the for loop together with the variables that are declared at a more global level. This local iterator is essentially a local variable declared in an open scope in which the body of the for loop is executed. The concept of closed scopes can be found in many modeling languages, such as Modelica, where interfaces specify the interaction points (ports) of components explicitly, and connectors are used to connect these ports. The ports and connectors can be mapped to the external variables and connect sets, respectively, of the CIF.

When scopes are nested, a variable used in an automaton binds to the declaration of that variable in the smallest enclosing scope that declares the variable. The search for this variable declaration starts in the smallest enclosing scope of the automaton. If the variable declaration is not found in an open scope, the search proceeds one level up in the scope hierarchy. If it is not found in a closed scope, the model is erroneous, because closed scopes may not have free variables. In all scopes (open and closed), inner scopes may redeclare variables that are already declared in outer scopes. The use of such a redeclared variable always refers to (binds to) its most local enclosing declaration.

By means of connect sets, variables in different scopes may be connected, so that the connected variables refer to the same variable. The following connections are possible:

- A connect set connecting external variables. The external variables may occur in nested scopes or in parallel scopes.
- A connect set connecting an internal variable to external variables. Such a connection is possible only in nested scopes, such that the internal variable is declared in the outer scope and the external variables are declared in, possibly parallel, inner scopes.

Internal variables cannot be connected to other internal variables. Action labels and channels can be connected to other action labels and channels, respectively, in a similar way as connections between variables are specified.

Chapter 8 presents several examples of the use of open and closed scopes, together with examples of the use of connect sets, internal and external variables and channels. By means of the keyword autoconnect, connections between external variables that have the same name are implicitly defined. Such a default corresponds to the semantics of languages such as PHAVER [18] and Charon [2].

## 7.2 Input and output variables

Input variables and output variables are special classes of external variable declarations. Input variables are by definition algebraic. The assumption for the input variables of an automaton is that their values are not restricted by the automaton in action nor in time transitions, in precisely the same way as required for the input variables in the automata of [1] and [17]. Note that this restriction is not enforced by the semantics of the CIF and neither by the semantics of the automata of [1] and [17]. The semantics does ensure that under the assumption that an automaton does not restrict the behavior of its own input variables, the input variables can also change arbitrarily in action and time transitions when that automaton is composed in parallel with another automaton, as long as the inputs are unconnected, or connected to other input variables only. In this way, when input variables are connected to an output variable, the behavior of the connection is completely determined by the output variable. Output variables cannot be connected to output variables of parallel automata. An output variable can be connected to an output variable of an outer block.

# 8 Example: Bottle filling system

The bottle filling system as shown in Figure 5 consists of a liquid storage tank, two identical bottle filling lines, and a bottle supply (see [31]).

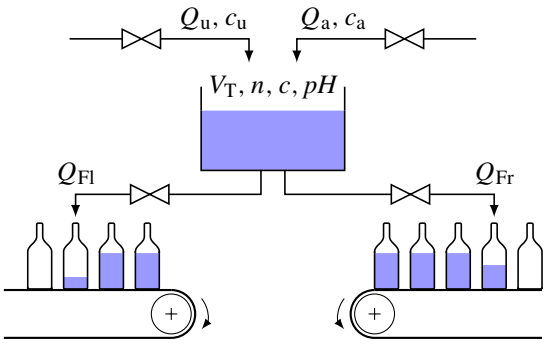


Figure 5: The bottle filling system.

The bottles are filled with liquid from the storage tank. A control system keeps the volume $V_T$ in the storage tank between 2 and 10, and the pH level (acidity) of the liquid in the storage tank between 7 and 7.1. The liquid in the storage tank slowly becomes less acidic (pH level increases). To correct this, a strong acid is dribbled into the storage tank when the acidity of the liquid becomes too low ($pH \geq 7.1$).

The acid and liquid supply processes are not modeled, since we consider the acid and liquid always to be available, and we are not interested in the amount of acid or liquid that is used.

The storage tank and the two bottle filling lines are connected by means of the variables $Q_{Fl}$, and $Q_{Fr}$, respectively. The storage tank is available in both bottle filling lines to prevent filling of the bottles when the storage tank is empty.

The molar quantity and molar concentration of the acid in the storage tank are denoted by $n$ and $c$, respectively, where $n = cV$. The incoming flows of liquid and acid of the liquid storage tank $T$ are denoted by $Q_u$ and $Q_a$, respectively. Acid leaves the tank in outgoing flows $Q_{Fl}$ and $Q_{Fr}$. The gradual reduction of the acidity of the liquid is modeled by means of a constant $K_{loss}$, which leads to

$$\dot{n} = c_u Q_u + c_a Q_a - c Q_{Fl} - c Q_{Fr} - K_{loss} V,$$

where $c_u$ and $c_a$ denote the concentrations of acid in the flows $Q_u$ and $Q_a$. Taking into account that the units of $c$ are in [mol/m$^3$] instead of [mol/l], the pH is given by

$$pH = -\log c/1000.$$

The behavior of the liquid storage tank is explained as follows. Initially, the pH of the liquid in the storage tank equals 7. It is assumed that the pH level of the incoming liquid is 7 or more, since the acidity controller can only make the acidity of the storage tank increase, causing the pH to decrease. If the pH value exceeds the maximum value (`pH >= 7.1`), the acid valve is opened (`alpha:= 1`) so that acid is dribbled into the tank. Dribbling of the acid continues until the pH value comes back at 7, and the valve is closed (`alpha:= 0`). In a similar way, the controller tries to keep the level of the storage tank between 2 and 10.

The behavior of the filling controller is explained as follows. When a new crate of bottles arrives, (`bottles?n`), where `n` denotes the number of bottles in a crate) the bottle volume is reset to 0, and the filling process and the bottle filling process is started (`VB,alpha := 0,1`). The valve switching the flow `QF` is modeled by means of the discrete variable `alpha`. Filling stops when the volume in the storage tank drops below 0.5 (`when VT <= 0.5 do alpha:= 0`). Filling resumes when the

volume in the storage tank is at least 0.7. Filling also stops when the bottle is full (`when VB >= 1 do alpha,n := 0,n-1`).

```
default edge urgent

model Bottle_Filling_System(val Qseta, Qsetu, ca, cu, Kloss: real) =
|[ connect {tank.V, left.VT, right.VT}
         , {tank.QFl, left.QF}
         , {tank.QFr, right.QF}
         , {bs.bottles, left.bottles, right.bottles}
:: tank:
     |[ output var V: cont real
      , extern var QFl, QFr: alg real
      , intern var alpha, beta: disc nat = (0,0)
                 , n: cont real
                 , pH: alg real = 7
                 , c, Qa, Qu: alg real
     :: |( mode physics =
             inv dot V = Qu + Qa - QFl - QFr
               , dot n = cu*Qu + ca*Qa - c*QFl
                         - c*QFr - Kloss*V
               ,     n = c*V
               ,    pH = - log(c)/1000
               ,    Qa = alpha*Qseta
               ,    Qu = beta*Qsetu
        :: physics
        )|
     || |( mode closed = when pH >= 7.1
                           do alpha := 1 to opened
             , opened = when pH <= 7
                           do alpha := 0 to closed
        :: closed
        )|
     || |( mode closed = when V <= 2
                           do beta := 1 to opened
             , opened = when V >= 10
                           do beta := 0 to closed
        :: closed
        )|
     ]|
|| left : Bottle_Filling_Line
|| right : Bottle_Filling_Line
|| bs : Bottle_Supply
]|

automaton Bottle_Filling_Line =
|[ input var VT: alg real
 , extern var QF: alg real
        , chan bottles?: nat
 , connect {fp.alpha, fc.alpha}
         , {fp.VB, fc.VB}
         , {bottles, fc.bottles}
:: fp : Filling_Physics
|| fc : Filling_Controller
]|

automaton Filling_Physics =
|[ input var alpha: nat
 , output var VB: cont real
 , extern var QF: alg real
:: |( mode m = inv dot VB = QF
                ,     QF = alpha*QsetF
   :: m
   )|
]|
```

```
automaton Filling_Controller =
|[ input var VB, VT: real
 , output var alpha: disc nat = 0
 , extern chan bottles?: nat
 , intern var n: disc nat = 0
:: |( mode start =
            when n = 0 act bottles?n
              do VB,alpha := 0,1 to filling
            when n > 0
              do VB,alpha := 0,1 to filling
        , filling =
            when VT <= 0.5
              do alpha := 0 to stopped
            when VB >= 1
              do alpha,n := 0,n-1 to start
        , stopped =
            when VT >= 0.7
              do alpha := 1 to filling
   :: start
   )|
]|

automaton Bottle_Supply =
|[ extern chan bottles!: nat
 , intern clock t
:: |( mode m = when t >= 2
                 act bottles!24
                 do t:= 0 to m
   :: m
   )|
]|
```

Figure 6 shows a graphical representation of (a part of) the CIF model of the bottle filling line. Solid (dashed) boxes represent closed (open) scopes, where the internal declarations are listed in the upper left corner, and the external declarations are represented as boxes (for variables) or triangles (for channels) on the borders of the box. Modes are visualized by means of circles, urgent (nonurgent) edges are represented as double (single) arrows between modes, and labeled with their guard, action, and update.

# 9 Mapping concrete syntax to abstract syntax

This section defines the meaning of a CIF model in the concrete syntax by means of a syntactical translation to the abstract format. First the concrete syntax is preprocessed as described in Section 9.1. The mapping of the concrete syntax to the abstract syntax is defined by means of function $\mathcal{T}$. It takes a preprocessed CIF model as input, and returns an automaton specified in the abstract syntax. This function is defined in Section 9.2.

## 9.1 Preprocessing

First, declarations of the form input var *varIds* : type or input var *varIds* : alg type are replaced by extern var *varIds* : alg type and declarations of the form output var *varDecls* are replaced by extern var *varDecls*. Then, all internal variables, clocks, channels and actions occurring in the body of the top-level closedScope are made unique. The external variable declarations are grouped w.r.t. their dynamic type into three different lists. The lists $discvars_{\text{ext}}$, $contvars_{\text{ext}}$, and $algvars_{\text{ext}}$ denote a comma seperated list of external variables declared with dynamic type disc, cont, and alg, respectively. Each entry in such a list is of the form $x : t = e$, where $x$ denotes a variable, $t$ denotes its static type, and $e$ denotes its initial value if the initial value is specified at the declaration. The declared external channels are grouped in list $chans_{\text{ext}}$, where each entry in

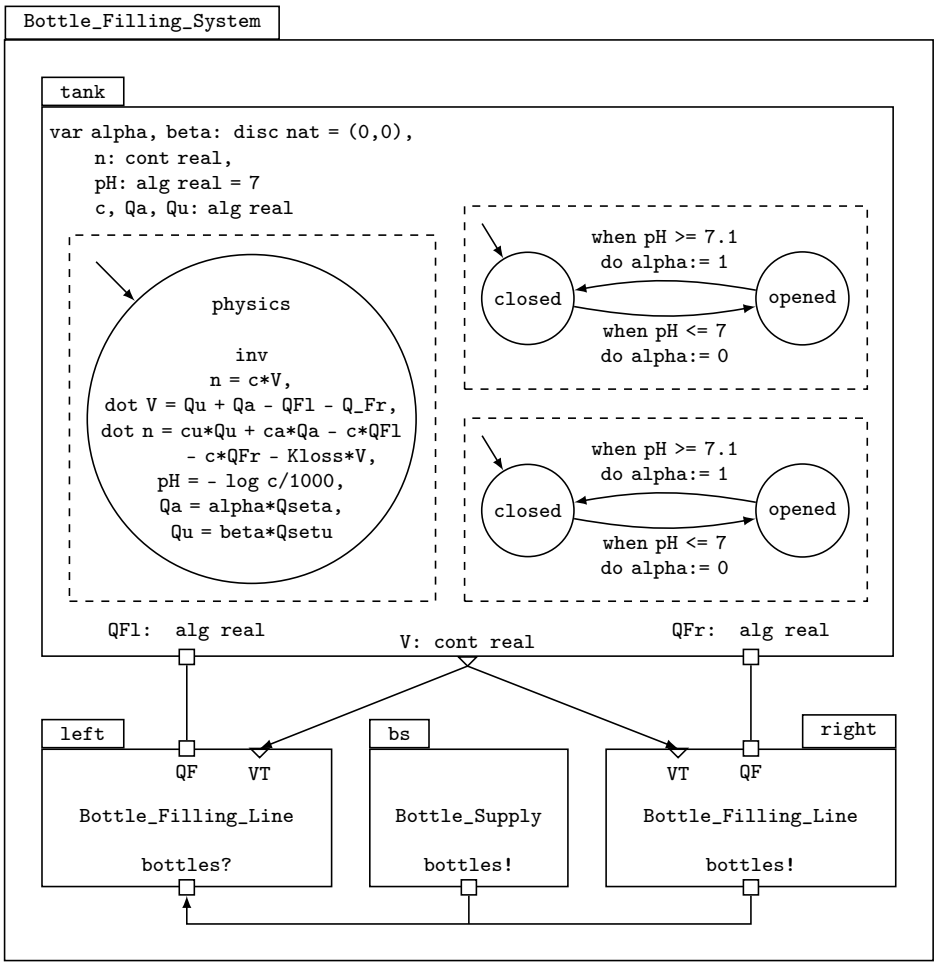


Figure 6: Graphical representation of the CIF model representing the bottle filling system.

the list is either of the form $c\,!:t$, $c\,?:t$, or $c:t$, where $c$ denotes a channel , and $t$ denotes its (static) type. Notation $\overline{x_0 : t_0 = e_0, \ldots, x_n : t_n = e_n}$ denotes the list $x_0, \ldots, x_n$, and notation $\overline{c_0 \Diamond_0 t_0, \ldots, c_n \Diamond_n t_n}$, where $\Diamond_i \in \{!:, ?:, :\}$ for $i \in \{0, \ldots, n\}$ denotes denotes the list $c_0, \ldots, c_n$. The external clocks and actions are grouped into respective lists $clocks_{\text{ext}}$, and $acts_{\text{ext}}$. A similar notation is used for all internal declarations. Finally, all process instantiations occurring in the model are flattened using the automaton definitions of the specification: Let

$$
\begin{array}{l}
l(p_0 : t_0, \ldots p_n : t_n) = \\
\text{|[ extern var } discvars_{\text{ext}}, contvars_{\text{ext}}, algvars_{\text{ext}} \\
\text{, clock } clocks_{\text{ext}}, \text{ chan } chans_{\text{ext}}, \text{ act } acts_{\text{ext}} \\
\text{, intern var } discvars_{\text{int}}, contvars_{\text{int}}, algvars_{\text{int}} \\
\text{, clock } clocks_{\text{int}}, \text{ chan } chans_{\text{int}}, \text{ act } acts_{\text{int}} \\
\text{, connect } set_1, \ldots, set_h \\
:: automaton \\
\text{]|}
\end{array}
$$

be the automaton definition associated with automaton definition identifier $l$, then automaton instantiation $l_{\text{a}} : l(e_0, \ldots e_n)$ denotes the following automaton:

$$\text{|[ extern var } discvars_{\text{ext}}, contvars_{\text{ext}}, algvars_{\text{ext}}$$

, clock $clocks_{\text{ext}}$, chan $chans_{\text{ext}}$, act $acts_{\text{ext}}$
, intern var $discvars_{\text{int}}$, $p_0 : t_0 = e_0, \ldots, p_n : t_n = e_n$,
, $contvars_{\text{int}}$, $algvars_{\text{int}}$
, clock $clocks_{\text{int}}$, chan $chans_{\text{int}}$, act $acts_{\text{int}}$
, connect $set_1, \ldots, set_h$
:: *automaton*
]‖

The parameter variables $p_0, \ldots, p_n$ of the automaton definition are declared as internal discrete variables inside the closedScope automaton, with initial values $e_0, \ldots, e_n$. It is assumed that the parameter variables and the external and internal variables and clocks from the closedScope are disjoint. Then, the possibly incomplete atomicAut constructs that occur in *automaton* are made complete according as follows.

By means of the default definitions *defaultDefs*, defaults can be specified for flow and urgency predicates. For edges, it can be specified whether edges are by default urgent (by means of keyword urgent) or not (by means of keyword nonurg). Similarly, for the jumping behavior of variables, by means of keyword jump the behavior of variables during an action transition as specified by an edge is such that they can change arbitrarily unless restricted by means of the update of that edge, or by means of keyword nonjump, the behavior of variables during an action transition as specified by an edge is such that they can remain unchanged unless a change in the values of the variables is explicitly restricted by means of the update of that edge.

An omitted default flow predicate denotes the default flow predicate true, and an omitted default tcp predicate denotes the default tcp predicate true. For edges, an omitted default urgency of an edge depends on the presence of an action specification (by means of the keyword act). If an action specification is present, the default urgency of the edge is nonurg, because action or channel labels should in principle be able to delay when communication or synchronization is not immediately possible. When there is no action specification, the default urgency of the edge is urgent. This is for example the case when the edge consists of an assignment only. Finally, an omitted default jumping specification of variables denotes the default nonjump.

Furthermore, by means of the keyword autoconnect, variables (or channels or actions) with the same name are connected.

According to the following defaults/transformations, the possibly incomplete atomicAut constructs that occur in *automaton* are made complete:

- An omitted invariant denotes the invariant true;
- An omitted flow denotes the default flow;
- An omitted tcp predicate denotes the default tcp predicate;
- An omitted guard denotes the guard true;
- For each urgent edge, the tcp predicate of the source mode is augmented with the conjunction of the negated guard of this edge. Then, the urgent keyword of the edge is replaced by keyword nonurg,
- An omitted label denotes the (non-synchronizing) label $\tau$;
- An update assignment $x := e$ is replaced by $\{x\} : x = e^-$, an omitted update denotes the empty update $\emptyset$ : true, i.e. the values of the discrete and continuous variables and the clocks remain unchanged.

## 9.2 Function $\mathcal{T}$

### Model

The translation of a (preprocessed) specification is defined as follows: $\mathcal{T}(\text{model } m = closedScope) = \mathcal{T}'_{(\emptyset,\emptyset,\emptyset,\emptyset,\text{true})}(closedScope)$.

Function $\mathcal{T}'$ takes two parameters: the first parameter contains variables, the dynamic type of variables, clocks, actions, and the initial values of variables, that are defined at a higher level than the second parameter. The second parameter contains elements of the preprocessed concrete syntax.

### Closed scope

A closed scope (BNF non-terminal *closedScope*) is mapped to an abstract CIF automaton as follows:

$$\begin{aligned}&\mathcal{T}'_{env}(\llbracket\ \mathsf{extern\ var}\ \mathit{discvars}_{\text{ext}}, \mathit{contvars}_{\text{ext}}, \mathit{algvars}_{\text{ext}}\\&\quad,\ \mathsf{clock}\ \mathit{clocks}_{\text{ext}}, \mathsf{chan\ urgent}\ \mathit{urgchans}_{\text{ext}}, \mathsf{chan\ nonurg}\ \mathit{nonurgchans}_{\text{ext}}\\&\quad,\ \mathsf{act\ urgent}\ \mathit{urgacts}_{\text{ext}}, \mathsf{act\ nonurg}\ \mathit{nonurgacts}_{\text{ext}}\\&\quad,\ \mathsf{intern\ var}\ \mathit{discvars}_{\text{int}}, \mathit{contvars}_{\text{int}}, \mathit{algvars}_{\text{int}}\\&\quad,\ \mathsf{clock}\ \mathit{clocks}_{\text{int}}, \mathsf{chan\ urgent}\ \mathit{urgchans}_{\text{int}}, \mathsf{chan\ nonurg}\ \mathit{nonurgchans}_{\text{int}}\\&\quad,\ \mathsf{act\ urgent}\ \mathit{urgacts}_{\text{int}}, \mathsf{act\ nonurg}\ \mathit{nonurgacts}_{\text{int}}\\&\quad,\ \mathsf{connect}\ \mathit{set}_1, \ldots, \mathit{set}_h\\&\quad::\ \mathit{automaton}\\&\quad\rrbracket\ ) =\\&\mathsf{hidevar}_{\overline{\mathit{vars}_{\text{int}}}}(\mathsf{hideact}_{\mathit{acts}_{\text{int}}, \mathit{chans}_{\text{int}}}(\mathsf{encap}_{\emptyset, \mathit{chans}_{\text{int}}}(\mathsf{urgent}_{\{\mathit{urgacts}_{\text{int}}\}, \{\mathit{urgchans}_{\text{int}}\}}(\mathcal{T}_{env'}(\mathit{automaton}'))))), \sigma_{\text{h}})\end{aligned}$$

where

- $\mathit{vars}_{\text{int}} = \{\mathit{discvars}_{\text{int}}, \mathit{contvars}_{\text{int}}, \mathit{algvars}_{\text{int}}\}$,
- $\mathit{env}' = (\overline{\mathit{vars}}, \mathit{dtype}, \mathit{acts}, \mathit{clocks}, \mathit{init})$,
- $\mathit{vars} = \mathit{vars}_{\text{ext}} \cup \mathit{vars}_{\text{int}}$,
- $\mathit{vars}_{\text{ext}} = \{\mathit{discvars}_{\text{ext}}, \mathit{contvars}_{\text{ext}}, \mathit{algvars}_{\text{ext}}\}$,
- $\mathit{dtype} = \{x \mapsto \mathsf{disc} \mid x \in \{\overline{\mathit{discvars}_{\text{ext}}}, \overline{\mathit{discvars}_{\text{int}}}\}\} \cup \{x \mapsto \mathsf{cont} \mid x \in \{\overline{\mathit{contvars}_{\text{ext}}}, \overline{\mathit{contvars}_{\text{int}}}\}\} \cup \{x \mapsto \mathsf{alg} \mid x \in \{\mathit{algvars}_{\text{ext}}, \mathit{algvars}_{\text{int}}\}\}$,
- $\mathit{acts} = \mathit{acts}_{\text{ext}} \cup \mathit{acts}_{\text{int}}$,
- $\mathit{acts}_{\text{ext}} = \{\mathit{urgacts}_{\text{ext}}\} \cup \{\mathit{nonurgacts}_{\text{ext}}\}$,
- $\mathit{acts}_{\text{int}} = \{\mathit{urgacts}_{\text{int}}\} \cup \{\mathit{nonurgacts}_{\text{int}}\}$,
- $\mathit{chans}_{\text{int}} = \{\mathit{urgchans}_{\text{int}}\} \cup \{\mathit{nonurgchans}_{\text{int}}\}$,
- $\mathit{clocks} = \{\mathit{clocks}_{\text{ext}}\} \cup \{\mathit{clocks}_{\text{int}}\}$,
- $\mathit{init} = \wedge_{x:\ x \in \overline{\mathit{vars}}, \text{value}_{\mathit{vars}}(x) \neq \bot}\ :\ x = \text{value}_{\mathit{vars}}(x)$,
- $\mathit{automaton}' = \mathit{automaton}[\mathit{setId}_1, \ldots, \mathit{setId}_h / \mathit{set}_1, \ldots, \mathit{set}_h]$,
- $\text{dom}(\sigma_{\text{h}}) = \emptyset$.

where function application $\text{value}_{\mathit{vars}}(x)$ returns the initial value for variable $x$ that is specified in *vars* or $\bot$ (undefinedness) otherwise. Notation $\mathit{automaton}[\mathit{setId}_1, \ldots, \mathit{setId}_h / \mathit{set}_1, \ldots, \mathit{set}_h]$ denotes the automaton where all free occurrences of identifiers from $\mathit{set}_i$ in *automaton* are replaced with unique identifier $\mathit{setId}_i$ for $i \in \{1, \ldots, h\}$.

Variables, clocks and actions occurring in *env* but not declared in the closedScope disappear. They are not used in the automaton. An automaton can be a closedScope, an atomicAut, an openScope, or a parallel composition of automata. In the next subsections, the mapping of the latter three is described.

### Atomic automaton

An atomic automaton (BNF non-terminal *atomicAut*), is mapped to an abstract CIF automaton as follows:

$$
\begin{array}{l}
\mathcal{T}'_{(vars,dtype,acts,clocks,init)}( \\
|( \text{ init } init_{\text{aut}} \\
\text{, mode} V_1 = \text{inv } i_1 \text{ flow } f_1 \text{ tcp } u_1 \\
\qquad\qquad \text{nonurg when } g_{1_1} \text{ act } a_{1_1} \text{ do } up_{1_1} \text{ to } V_{1_1} \\
\qquad\qquad \vdots \\
\qquad\qquad \text{nonurg when } g_{1_{k_1}} \text{ act } a_{1_{k_1}} \text{ do } up_{1_{k_1}} \text{ to } V_{1_{k_1}} \\
\vdots \\
\text{, mode} V_n = \text{inv } i_n \text{flow } f_n \text{tcp } u_n \\
\qquad\qquad \text{nonurg when } g_{n_1} \text{ act } a_{n_1} \text{ do } up_{n_1} \text{ to } V_{n_1} \\
\qquad\qquad \vdots \\
\qquad\qquad \text{nonurg when } g_{n_{k_n}} \text{ act } a_{n_{k_n}} \text{ do } up_{n_{k_n}} \text{ to } V_{n_{k_n}} \\
:: v_0 \\
)|) = (X, \emptyset, dtype, V, v_0, \text{init}, \text{flow}, \text{inv}, \text{tcp}, acts, E)
\end{array}
$$

where

- $X = vars \cup clocks$,
- $V = \{V_1, \ldots, V_n\}$,
- $\text{init} = init \wedge init_{\text{aut}} \wedge (\wedge_{x:x \in clocks} : \ x = 0)$,
- $\text{dom}(\text{flow}) = \text{dom}(\text{inv}) = \text{dom}(\text{tcp}) = V$,
  $\forall_{i:i \in \{1,\ldots,n\}} : \ \text{flow}(V_i) = f_i \wedge (\wedge_{x:x \in clocks} : \ \dot{x} = 1)$,
  $\text{inv}(V_i) = i_i$,
  $\text{tcp}(V_i) = u_i$,
- $E = \{(V_i, g_{i_j}, a_{i_j}, up_{i_j}, V_{i_j}) \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}\}$.

### Open scope

An open scope (BNF non-terminal *openScope*) is mapped to an abstract CIF automaton as follows: Let $env = (vars, dtype, acts, clocks, init)$, then

$$
\begin{array}{l}
\mathcal{T}'_{env}(|( \text{ extern var } discvars_{\text{ext}}, contvars_{\text{ext}}, algvars_{\text{ext}} \\
\qquad \text{, clock } clocks_{\text{ext}}, \text{ chan urgent } urgchans_{\text{ext}}, \text{ chan nonurg } nonurgchans_{\text{ext}} \\
\qquad \text{, act urgent } urgacts_{\text{ext}}, \text{ act nonurg } nonurgacts_{\text{ext}} \\
\qquad \text{, intern var } discvars_{\text{int}}, contvars_{\text{int}}, algvars_{\text{int}} \\
\qquad \text{, clock } clocks_{\text{int}}, \text{ chan urgent } urgchans_{\text{int}}, \text{ chan nonurg } nonurgchans_{\text{int}} \\
\qquad \text{, act urgent } urgacts_{\text{int}}, \text{ act nonurg } nonurgacts_{\text{ext}} \\
\qquad \text{, connect } set_1, \ldots, set_h \\
\qquad :: automaton \\
\qquad )| \ ) = \\
\text{hidevar}_{\overline{vars_{\text{int}}}}(\text{hideact}_{acts_{\text{int}}, chans_{\text{int}}}(\text{encap}_{\emptyset, chans_{\text{int}}}(\text{urgent}_{\{urgacts_{\text{int}}\}, \{urgchans_{\text{int}}\}}(\mathcal{T}_{env'}(automaton')), \sigma_{\text{h}})
\end{array}
$$

where

- $vars_{\text{int}} = \{discvars_{\text{int}}, contvars_{\text{int}}, algvars_{\text{int}}\}$,

- $env' = (\overline{vars'}, dtype', acts', clocks', init')$,
- $vars' = vars \cup \overline{vars_{\text{int}}}$,
- $dtype' = dtype \cup \{x \mapsto \mathsf{disc} \mid x \in \{\overline{discvars}_{\text{int}}\}\} \cup \{x \mapsto \mathsf{cont} \mid x \in \{\overline{contvars}_{\text{int}}\}\} \cup \{x \mapsto \mathsf{alg} \mid x \in \{\overline{algvars}_{\text{int}}\}\}$,
- $acts' = acts \cup acts_{\text{int}}$,
- $acts_{\text{int}} = \{urgacts_{\text{int}}\} \cup \{nonurgacts_{\text{int}}\}$,
- $chans_{\text{int}} = \{urgchans_{\text{int}}\} \cup \{nonurgchans_{\text{int}}\}$,
- $clocks' = clocks \cup \{clocks_{\text{int}}\}$,
- $init = init \wedge (\wedge_{x:\ x \in \overline{vars}_{\text{int}}, \text{value}_{vars_{\text{int}}}(x) \neq \bot} :\ x = \text{value}_{vars_{\text{int}}}(x))$,
- $automaton' = automaton[setId_1, \ldots, setId_h / set_1, \ldots, set_h]$,
- $\text{dom}(\sigma_{\text{h}}) = \emptyset$.

### Parallel composition

Function $\mathcal{T}'_{env}$ distributes over parallel composition: $\mathcal{T}'_{env}(automaton_{\text{l}}) \parallel automaton_{\text{r}}) = \mathcal{T}'_{env}(automaton_{\text{l}}) \parallel \mathcal{T}'_{env}(automaton_{\text{r}})$

### Urgent operator

The order of the application of function $\mathcal{T}'_{env}$ and the urgent operator to an automaton is irrelevant: $\mathcal{T}'_{env}(\text{urgent}_{L_{\text{u}}, H_{\text{u}}}(automaton)) = \text{urgent}_{L_{\text{u}}, H_{\text{u}}}(\mathcal{T}'_{env}(automaton))$

### Encapsulation operator

The order of the application of function $\mathcal{T}'_{env}$ and the encapsulation operator to an automaton is irrelevant: $\mathcal{T}'_{env}(\text{urgent}_{L_{\text{u}}, H_{\text{u}}}(automaton)) = \text{urgent}_{L_{\text{u}}, H_{\text{u}}}(\mathcal{T}'_{env}(automaton))$

# 10 Concluding remarks

The proposed interchange automaton format integrates formalisms rooted in computer science with those rooted in dynamics and control. It incorporates the major building blocks required for hybrid system specification. Future work entails, among others, extending the format with OR-super states, such as defined in [21], possibly extending the interchange format with stochastic model primitives, and defining the syntax of the transfer interchange formats. The development of translations and simulator implementations will be done by different partners in Work Package 3 of the HYCON NoE [27].

The atomic interchange automata as introduced syntactically in Definition 4.2 and for which the semantics has been introduced in Section 5 are very expressive. For any application of an action or variable hiding operator on an atomic interchange automaton, it is possible to obtain an equivalent atomic interchange automaton. Also, the parallel composition of any two atomic interchange automata for which the shared variables have compatible types can be replaced by an equivalent atomic interchange automaton. The only operator that cannot be eliminated in all relevant cases is the urgent action operator.

# Bibliography


[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[2] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceeddings of the IEEE*, 91(1):11–28, January 2003.

[3] R. Alur, T. A. Henzinger, and P. H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[4] Rajeev Alur, Radu Grosub, Insup Lee, and Oleg Sokolsky. Compositional modeling and refinement for hierarchical hybrid systems. *Journal of Logic and Algebraic Programming*, 68(1-2):105–128, 2006.

[5] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129–210, 2006.

[6] D. A. van Beek, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Foundations of an interchange format for hybrid systems. In *Hybrid Systems: Computation and Control, 10th International Workshop*, Lecture Notes in Computer Science, Pisa, 2007. Springer-Verlag. to be published.

[7] D. A. van Beek, M. A. Reniers, R. R. H. Schiffelers, and J. E. Rooda. Foundations of a compositional interchange format for hybrid systems. Technical Report SE-Report 2006-05, Eindhoven University of Technology, Department of Mechanical Engineering, The Netherlands, 2006. http://se.wtb.tue.nl/sereports/.

[8] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceeddings of the IEEE*, 91(1):64–83, January 2003.

[9] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. Modest: A compositional modeling formalism for real-time and stochastic systems. Technical Report Technical Report TR-CTIT-04-46, University of Twente, Centre for Telematics and Information Technology, The Netherlands, 2004.

[10] Tommaso Bolognesi and Ferdinando Lucidi. Timed process algebras with urgent interactions and a unique powerful binary operator. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice, REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 124–148, Mook, The Netherlands, 1991. Springer-Verlag.

[11] Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Information and Computation*, 163(1):172–202, 2000.

[12] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[13] Stefano Di Cairano, Alberto Bemporad, and Michal Kvasnica. An architecture for data interchange of switched linear systems. Technical Report D 3.3.1, HYCON NoE, 2006.

[14] Columbus IST project. http://www.columbus.gr, 2006.

[15] EcosimPro. http://www.ecosimpro.com, 2006.

[16] A. F. Filippov. *Differential Equations with Discontinuous Right Hand Sides*. Kluwer Academic Publishers, Dordrecht, 1988.

[17] G. Frehse. *Compositional verification of hybrid systems using simulation relations*. PhD thesis, Radbout University Nijmegen, 2004.

[18] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer-Verlag, 2005.

[19] Biniam Gebremichael and Frits Vaandrager. Specifying urgency in timed i/o automata. In *Third IEEE Conference on Sofware Engineering and Formal Methods*, pages 64–74, 2005.

[20] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[21] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[22] W. P. M. H. Heemels, B. de Schutter, and A. Bemporad. Equivalence of hybrid dynamical models. *Automatica*, 37(7):1085–1091, 2001.

[23] T. A. Henzinger. Masaccio: A formal model for embedded components. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Sendai, Japan, August 17-19, 2000, Proceedings*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer-Verlag, 2000.

[24] T. A. Henzinger. The theory of hybrid automata. In M.K. Inan and R.P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series F: Computer and Systems Science*, pages 265–292. Springer-Verlag, New York, 2000.

[25] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to HYTECH. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71, Aarhus, Denmark, 1995. Springer-Verlag.

[26] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood-Cliffs, 1985.

[27] HYCON Network of Excellence. http://www.ist-hycon.org/, 2005.

[28] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium*, volume 571 of *Lecture Notes in Computer Science*, pages 591–620, Nijmegen, The Netherlands, 1992. Springer-Verlag.

[29] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[30] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[31] K. L. Man and R. R. H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. PhD thesis, Eindhoven University of Technology, 2006.

[32] Sven Erik Mattsson, Martin Otter, and Hilding Elmqvist. Modelica hybrid modeling and efficient simulation. In *38th IEEE Conference on Decision and Control*, pages 3502–3507, 1999.

[33] MoBIES team. HSIF semantics. Technical report, University of Pennsylvania, 2002. internal document.

[34] M. R. Mousavi, M. A. Reniers, and J. F. Groote. Notions of bisimulation and congruence formats for SOS with data. *Information and Computation*, 200(1):107–147, 2005.

[35] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer, 1993.

[36] Alessandro Pinto, Luca P. Carloni, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Interchange format for hybrid systems: Abstract semantics. In João P. Hespanha and Ashish Tiwari, editors, *Hybrid Systems: Computation and Control, 9th International Workshop*, volume 3927 of *Lecture Notes in Computer Science*, pages 491–506, Santa Barbara, 2006. Springer-Verlag.

[37] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[38] J.W. Polderman and J.C. Willems. *Introduction to Mathematical Systems Theory: A Behavioral Approach*. Springer-Verlag, New York, 1998.

[39] A. J. van der Schaft and J. M. Schumacher. *An Introduction to Hybrid Dynamical Systems*, volume 251 of *Lecture Notes in Control and Information Sciences*. Springer-Verlag, 2000.

[40] The MathWorks, Inc. *Using Simulink, version 6*. http://www.mathworks.com, 2005.

[41] Michael Tiller. *Introduction to Physical Modeling with Modelica*, volume 615 of *The International Series in Engineering and Computer Science*. Springer-Verlag, 2001.

[42] Uppsala University and Aalborg University. *UPPAAL version 4.0 online help*, 2006.

[43] Michael von der Beeck. A comparison of statecharts variants. In Hans Langmaack, Willem P. de Roever, and Jan Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, Lübeck, Germany, 1994. Springer-Verlag.